# VisualAge TeamConnection Version 3: Simple scenario using the build function in AIX

Document Number TR 29.3098

Angel Rivera, Michael Petersen, Dev Banerjee

VisualAge TeamConnection and CMVC Development
IBM Software Solutions
Research Triangle Park, North Carolina

# ABSTRACT

This technical report describes how to use the build function of VisualAge TeamConnection Version 3 for developing a simple build scenario in Unix, showing every single step in detail.

The main objective is to fill the gap that exists in the existing documentation for the product and to provide in a single place some concrete steps and suggestions for using this function in a simple scenario.

Because this is a simple build scenario, only the main concepts will be described; advanced techniques will not be examined, such as performance issues.

The target audience for this technical report are build administrators and software developers who are familiar with VisualAge TeamConnection and with the process of building an application.

## ITIRC KEYWORDS

- VisualAge TeamConnection
- Build function
- Build script
- Unix
- AIX

# ABOUT THE AUTHORS

## ANGEL RIVERA

Mr. Rivera is a Advisory Software Engineer in the VisualAge TeamConnection and CMVC development group. He joined IBM in 1989 and since then has worked in the development and support of library systems.

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 Tecnológico y de Estudios Superiores de Monterrey, México.

## MICHAEL PETERSEN

Mike Petersen is an IBM software developer for service offerings at Research Triangle Park, North Carolina. Mike has worked in systems, applications, and other product programming areas. He has also has assignments with world trade, IBM faculty loan program, technical education, and technical support for marketing.

Mike received a BS in mathematics with a physics minor in 1972 and an MS in computer science in 1974 from Purdue University. In 1979 he received an MBA from Marist College.

## DEV BANERJEE

Dev Banerjee is an IBM software developer at Research Triangle Park, North Carolina. He is currently with Visualage TeamConnection Development and has been with IBM since 1984.

Dev received a MS in physics from Indian Institute of Technology, Kanpur, India, in 1977 and an MS in Systems Science in 1979 from LSU, Baton Rouge, Lousiana.

# CONTENTS

# FIGURES

# INTRODUCTION

This technical report describes how to use the Build function of VisualAge TeamConnection Version 3 for developing a simple build scenario in AIX, showing every single step in detail. Although the chosen platform is AIX, the main concepts are the same for the other Unix and Intel platforms.

The main objective is to fill the gap that exists in the existing documentation for the product and to provide in a single place some concrete steps and suggestions for using this function in a simple scenario.

Because this is a simple build scenario, only the main concepts will be described; advanced techniques will not be examined, such as performance issues.

The target audience for this technical report are build administrators and software developers who are familiar with VisualAge TeamConnection and with the process of building an application.

This technical report used the fixpak 3.0.1 of VisualAge TeamConnection.

## ACKNOWLEDGEMENTS

Many of the questions and answers that are compiled in this technical report were obtained from the TEAMC and CMVC forums in the IBMPC conferencing disk and from the CMVC6000 forum in the IBMUNIX conferencing disk.

We want to thank the main participants in these electronic forums for their support, especially to:

- Lee Perlov, from Customer Services of the VisualAge TeamConnection development team, IBM Software Solutions, RTP, North Carolina, USA.
- Jeff Hook, from Customer Services of the VisualAge TeamConnection development team, IBM Software Solutions, RTP, North Carolina, USA. He provided very useful hints on using the Build function.
- Tim Orlowski, Service Support of the VisualAge TeamConnection development team, IBM Software Solutions, RTP, North Carolina, USA.
- Jose Maria Rodriguez, IBM Games Systems Center, Madrid, Spain.

## GETTING THIS TECHNICAL REPORT

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

- Library home page by selecting the item Library at URL:

  **http://www.software.ibm.com/ad/teamcon**

- FTP site by accessing the URL:

  **ftp://ftp.software.ibm.com/ps/products/teamconnection/papers**

  See the file README.index.txt for details.

## COMMENTS ABOUT THE FORMATTING OF LINE COMMANDS IN THIS DOCUMENT

All the TeamConnection client actions described in this technical report were done thru the Unix GUI. However, in order to document the actual line command that was used (just in case you need to use them in a shell script), after each command that was executed thru the GUI, we looked at the bottom of the Log file teamc.log file (specified in the Setup page in the Settings) to copy the command that the GUI prepared using the input from the user and that was sent to the family server. Then we pasted that command in this document.

Furthermore, the GUI sometimes uses input parameters that are optional such as the "-type TCPart" in the "teamc Part -build" command. But because we did a copy and paste operation, and we want to report what we saw during our exercises, we did not delete any input arguments.

The only editing that we did was to split the lines in order to fit them within the page width of this document. We had to split the very long line commands by using the '\' character. Thus, when you see a command that takes 3 lines, you should really think of it as being in one single line.

## PREREQUISITES FOR RUNNING THE EXAMPLES IN THIS TECHNICAL REPORT

The following are the software prerequisites for running the examples in this technical report:

- VisualAge TeamConnection Enterprise Server Version 3 and DB2 Universal Database Version 5.

  They are used for handling the TeamConnection family which will manage the code to be built.

**Note:** This technical report used the fixpak 3.0.1 of VisualAge TeamConnection.

- A C compiler, such as "xlc", "xlC" or "cc". In this TR, the "xlc" compiler will be used. To find out if you have it, you can issue the following commands in AIX:
  - To find out the location (default):

    ```
    $ which xlc
    /usr/bin/xlc
    ```

  - To find out the version:

    ```
    $ lslpp -al | grep -i xlC.C | more
    ```

    One of the lines in the output should say:

    ```
     xlC.C                       3.1.4.0  COMMITTED  C for AIX Compiler
    ```

- 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.

    A zip file prepared in Unix can be unzipped in the correct format in Windows NT and vice versa. This is something that the other ZIP programs such as "gnuzip" or "gzip" cannot do.

  Info-ZIP's software (zip, unzip and related utilities) is free and can be obtained for the desired platforms from various anonymous-ftp sites, including the URL:

  `ftp://ftp.uu.net:/pub/archiving/zip/`

# SIMPLE (BUT COMPLETE) SCENARIO

The following simple scenario shows the main type of build events that a developer could have and how they are mapped into the Build function of VisualAge TeamConnection.

## DEVELOPER'S VIEW OF THE BUILD EVENTS AND THE RELATED PARTS

In AIX, a C source code file (archivo.c) which includes a header file (archivo.h) will be compiled producing the object code file (archivo.o), which in turn will be linked to produce the executable file (archivo).

This executable file will be packed with the release notes (readme.txt) into a zip file (archivo.zip) which is going to be delivered to other people.

Finally, a collector file (archivo.col) will be used to ensure that the developer's notes (devnotes.txt) are in sync with the zip file, even though these notes are not going to be given to others.

**Language trivia:** By the way, the word "archivo" means "file" in Spanish for Mexico and most Latin America, although the term "fichero" is preferred in Spain.

Before the VisualAge TeamConnection build tree is presented, Figure 1 on page 6 shows all parts and build events that are involved. The sequence is shown from top to bottom, using the file extensions (suffixes) that are typical in a Unix environment.

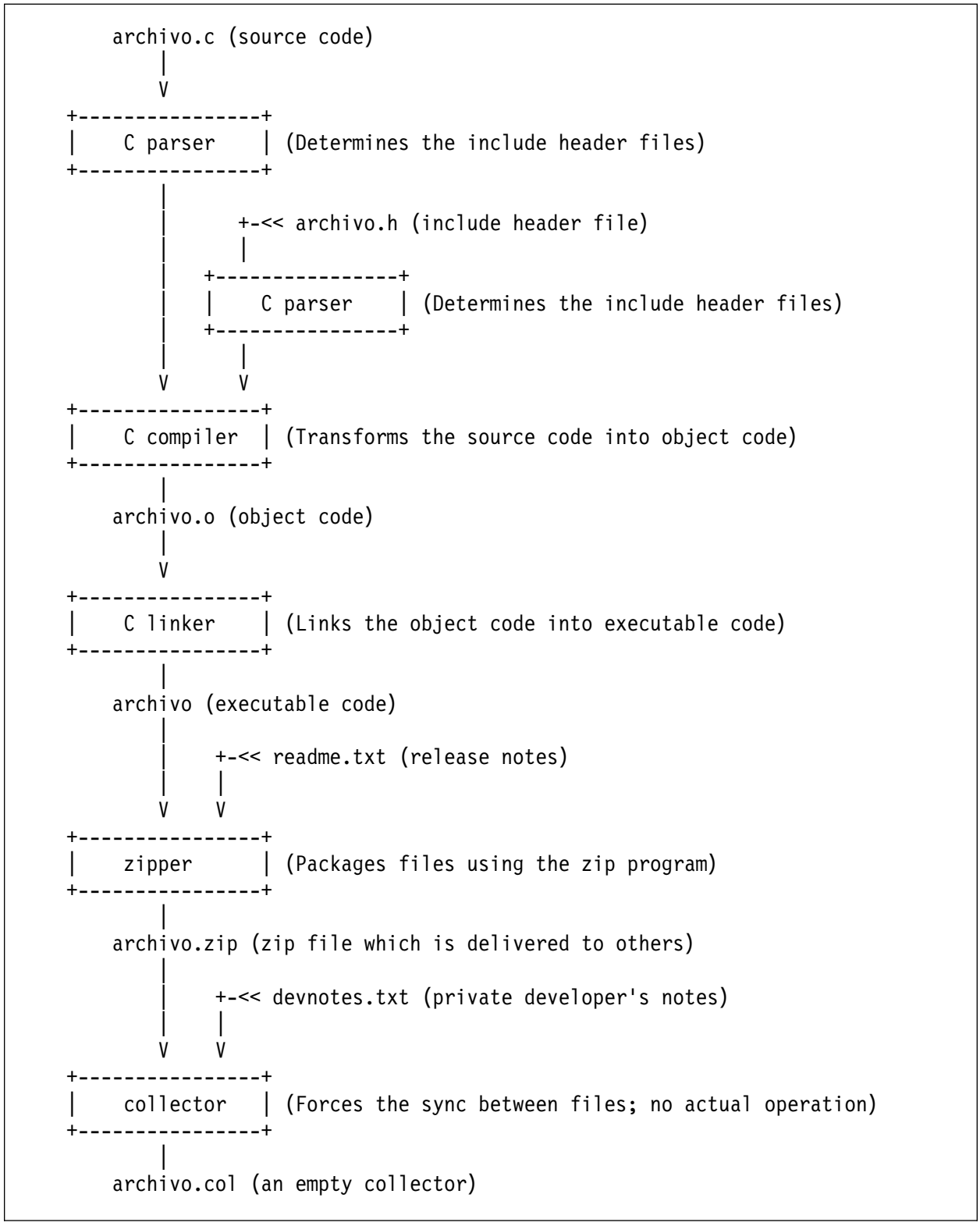



**Figure 1. Developer's view of the build events and the related parts**

The collector file is empty and the only function is to sync other files, similar to a target name in a makefile.

Notice that this view is "tool-centric". It is centered along the transformational tools such as the C compiler and, by the way, it shows the parts that are involved.

## TRANSFORMING THE DEVELOPER'S VIEW INTO THE TEAMCONNECTION BUILD VIEW

From this developer's view of the scenario, let us see an intermediate mapping to gradually change the representation of the build events and the associated parts into something to which we can easily relate when using the Build function.

First of all, in the Build function in VisualAge TeamConnection the build events are not stand-alone. They are always associated with a given file or set of files. Specifically, these build events are associated with the output of the builder (transformational tool) and not with the inputs to the builder. For example, the C compiler build event is associated with the object code file, which is the output of the builder which is a C compiler; the build event is not associated with the source code or the include header file.

The exception to this rule is the parser, which does not really have an output file; rather, it parses the source code and determines the include files which are dependencies to the source code. In this case, the parser is associated with the source code.

In order to better show the mapping between the developer's view of the build events and the Build function view, let us transform the Figure 1 on page 6 into an intermediate view. Figure 2 on page 8 shows the build events associated with the corresponding output parts.

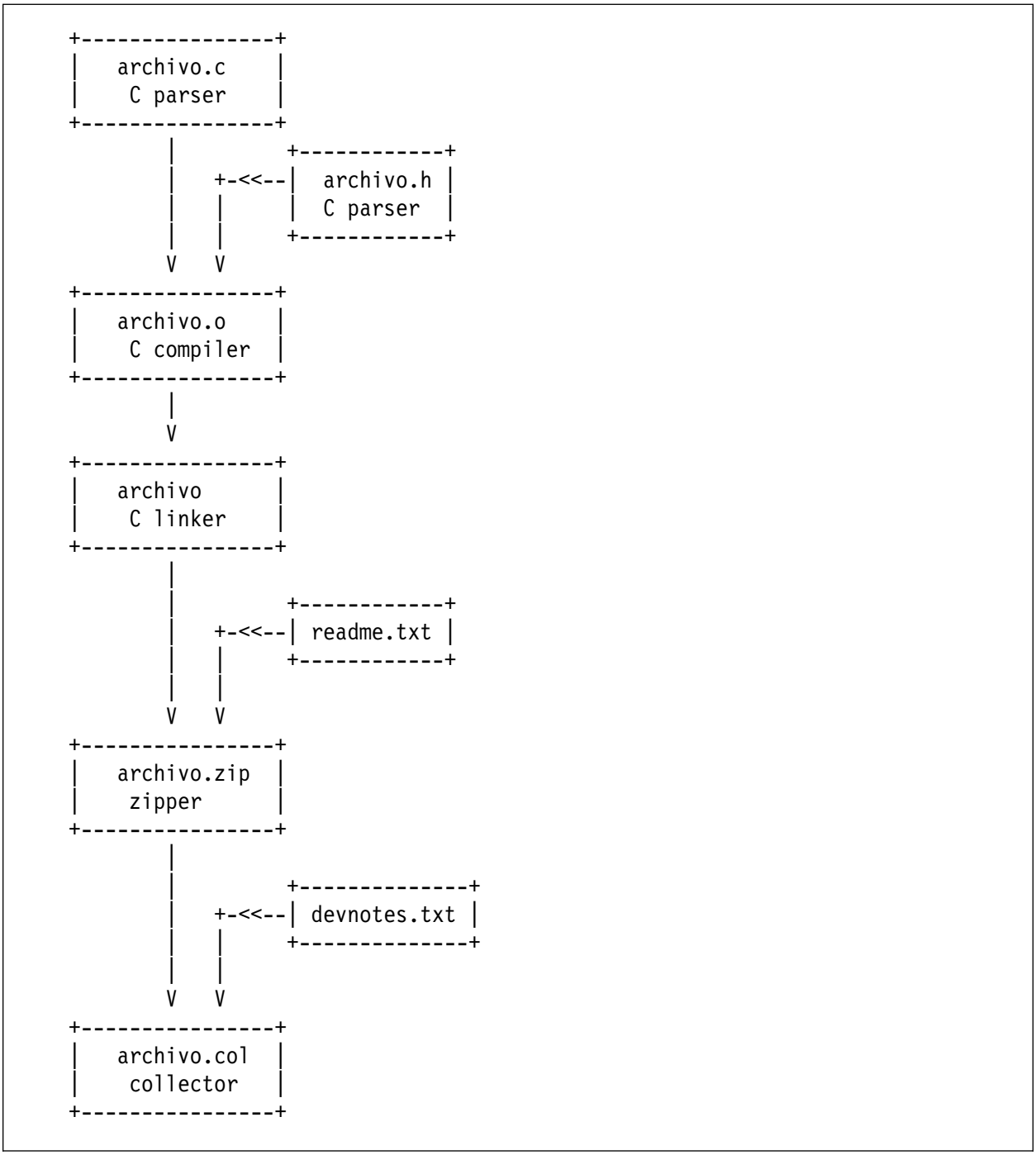



**Figure 2. Intermediate mapping view**

Notice that there is a big change in the representation of the build events and the related parts: this view is "part-centric", that is, the parts are now in the center, and by the way, the figure shows the tool that is associated with the part (such as the parser for the source code) or the tool that produces the part (such as the C compiler for the object code).

Notice also that there are some parts that are not related to any tool, such as the release notes (readme.txt). They are not parsed and they are not the output of an automated transformational tool (although the transformational tool is the text editor, which is used by the user to transform the ideas on the user's head into editable text, but this is not a process that can be automated!)

## TEAMCONNECTION BUILD VIEW OF THE BUILD EVENTS AND THE RELATED PARTS

From the intermediate view of the scenario, let us see how these build events map to the Build function in VisualAge TeamConnection.

First of all, we have to reverse the build tree and show it up-side down! That is, instead of showing first (at the top) what are the input files and then showing (at the bottom) what are the output files (as in the developer's view), the build tree for the Build function shows first the output files.

Why does the user want to see the output files first? Since the Build function is part-centric and not tool-centric, the user can select the desired part and select the build action, and the Build function will see if this part is up-to-date with respect to the dependencies; if affirmative, then there is nothing to do; if negative (that is, the part is out-of-date) then the transformational tool needs to be invoked using the dependency parts. However, this process is recursive: when a dependency part is involved, the Build function determines also if the part is up-to-date, if not, it then invokes the transformational tool for that part, and so on.

The beauty of this approach is that the user can select the top build part (in this case the collector file) and request the build action, which will trigger a cascade of build events down to the last involved part. That is the reason why this collector part is now shown at the TOP of the Build view in Figure 3 on page 10.

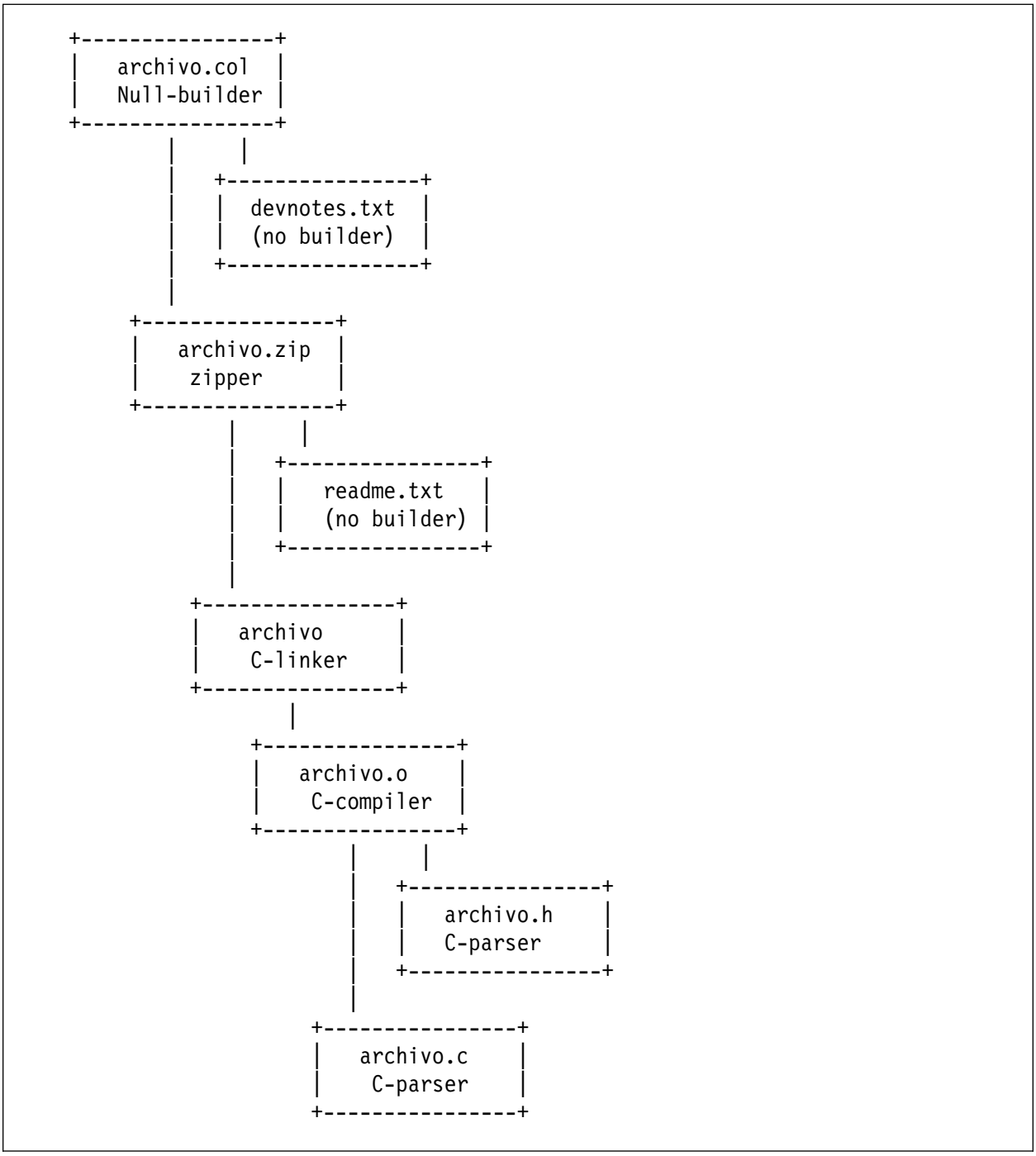



**Figure 3. Build view of the build events and the related parts**

Notice these other changes in the representation, in order to better show what you will see when using the BuildView in the TeamConnection GUI:

- The top build node part (collector file) is shown now at the top, and in the left-most position.

- The dependencies (input parts) are shown indented to the right with respect to the output part.
- The input parts are shown below the output parts.
- The transformational tools or **builders** are now represented with the actual name in which they will be known in TeamConnection. For example, the C compiler will be "C-compiler".
  - Notice that the "Null-builder" is a builder that does nothing (a NO-OP or no-operation). However, it still requires a build server to be running.
  - Notice that the term "no builder" is not really a builder, that is, the part is not associated with any builder. In other words, when creating this part in TeamConnection the Builder field should be left blank.
  - Do not use the name "null" for a Builder name because "null" is a reserved word in TeamConnection.
- Each part may have an associated builder or parser to it.
  - The header file can have also other include files. Thus, it is associated with the C-parser.
  - The text files are not associated with any transformational tools, thus we have to say it explicitly by not assigning any builders to them; thus, the term "no builder".

Finally, a clarification of terminology: in the Build function, a "parent part" is just an output part, and a "child part" is just an input part. In this case, the collector part (archivo.col) is the parent of the developer's notes (devnotes.txt) and the zip file (archivo.zip) which are the children parts. This relationship is propagated to the other parts, for example, the executable file archivo has two roles: it is a child for the collector part archivo.col and it is a parent for the object file archivo.o.

## REPRESENTATION OF THE BUILD TREE IN THE BUILDVIEW WINDOW

The conceptual build tree shown in Figure 3 on page 10 is actually represented in Figure 4 on page 12 when using the BuildView window of the TeamConnection GUI in Windows NT:

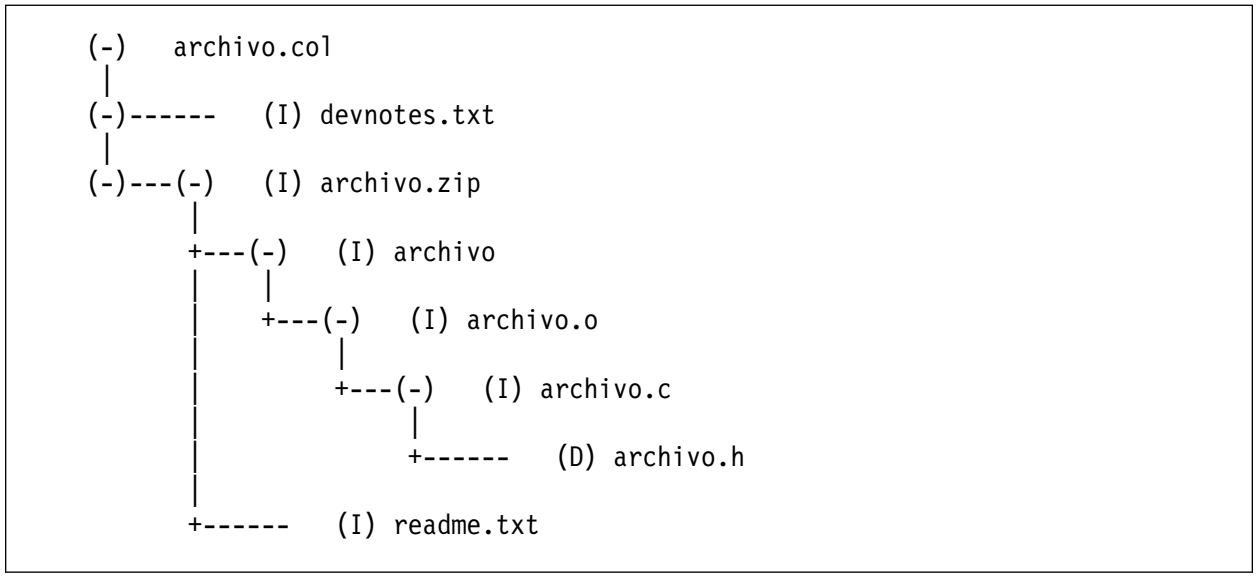



**Figure 4. Using BuildView from the Windows NT GUI to represent the build tree**

There is a bug with the Unix GUI (hopefully to be fixed after fixpak 3.0.1), in which the build tree is shown incorrectly (the readme.txt is shown as input to the object code archivo.o instead of input to the zip file archivo.zip):

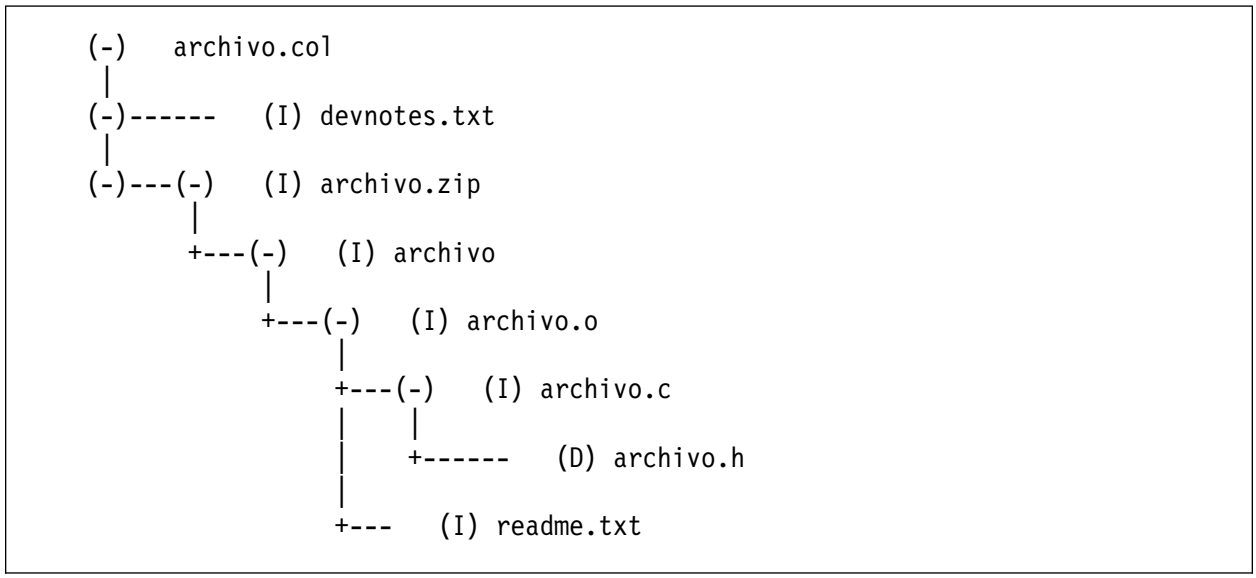



**Figure 5. Using BuildView from the Unix GUI to represent the build tree**

**Notes:**

1. The items identified with **(I)** are "InputTo" the part that is shown above them. For example, archivo.c is an input to archivo.o.
2. The items identified with **(D)** are "DependsOn" the part that is shown above them. For example, archivo.c depends on the header file archivo.h.
3. There is a bug in the Unix GUI and in the Windows GUI v300 when using passwords that prevents the '+' signs to be expanded and thus, the true "tree" view is not possible in v300.

   This bug was fixed in v301 for the Unix GUI

   This bug is still present in the Windows 95 version of the GUI v301 when using password. This bug is NOT present in the Windows NT version of the GUI v301.
4. In fixpak 3.0.1 of the Unix GUI, after you click on the '+' sign for the first time, you will see that the progress window is shown and then the display will be refreshed. Notice that '+' sign disappears without expanding the build tree.

   You need to click again on the empty space where the '+' sign was in order to expand the build tree.

# SETUP ACTIVITIES

This chapter describes the steps involved in the preparation for the build activities.

## PRELIMINARY TASKS

- It is necessary to have an existing VisualAge TeamConnection Version 3 family. In this document this family will have the following characteristics:
  - family name (TC_FAMILY): "tcpc3@oemppc3@3420"
  - superuser: "tcpc3"
  - host list entry for the superuser: TeamConnection user id "tcpc3", login as "tcpc3" from host "oem-ppc3"
  - normal user: "rivera"
  - host list entry for the normal user: TeamConnection user id "rivera", login as "tcpc3" from host "oem-ppc3"
  - authentication level: "PASSWORD_OR_HOST"
- The .profile for the family must be customized to your local environment. In this example, the following environment variables were specified:

```
export TC_FAMILY="tcpc3@oemppc3@3420"
export TC_BECOME=rivera
export TC_TOP=/home/tcpc3/work

export LIBPATH=/usr/lib:$TC_HOME/lib
```

**Notes:**

1. The environment variable TC_FAMILY indicates the TeamConnection family to be used. The syntax is: familyName@hostName@portNumber
2. The environment variable TC_BECOME indicates the TeamConnection user id (in contrast to the System Login to the operating system).
3. The environment variable TC_TOP indicates the top directory that TeamConnection will use for extracting, creating, checking out and checking in parts.
4. In order to run the TeamConnection commands, it is necessary to add $TC_HOME/lib to the LIBPATH; if this is not done, then the following error message will be displayed when entering a TeamConnection command such as "teamc":

```
exec(): 0509-036 Cannot load program teamc because of the following errors:
        0509-022 Cannot load library libfhccmnc.a.
        0509-026 System error: A file or directory in the path name does not ex
ist.
```

5. If LIBPATH is not specified at all, or if /usr/lib is not specified, then when running the executable source code generated by this example, you may get the following error message:

```
exec(): 0509-036 Cannot load program archivo because of the following errors:
        0509-022 Cannot load library libc.a[shr.o].
        0509-026 System error: A file or directory in the path name does not ex
ist.
```

## SETUP OF THE FAMILY AND BUILD SERVER

In this section, the family server, the notification daemon, the build server and the GUI will be started.

Even though in this example, the family administrator sets up and runs the build server on the family server machine, the build server is not restricted to be run this way. Specifically:

- The build server can be run on any other machine where the client is installed.
- It can be started by any TeamConnection user who can issue teamc client commands from that machine.
- The family server uses usual authentication criteria before allowing the build server to access the family.

The setup sequence is the following:

1. Login as the family administrator. In this example it is called: tcpc3
2. Open a window that will be dedicated to the build server. The build server will display messages in this window and you cannot use this window for anything else.

   Specify the environment (flag -e) "AIX" and the pool (flag -p) "pool1", which are values that serve as identifiers which are used by the build function. You can use any values that you want, as long as they are used consistently.

   It is recommended also to use the flag -s to send the output to the screen. In any case the message are also stored in $HOME/teamcbld.log.

   ```
   teamcbld -e AIX -p pool1 -s
   ```

   You will see the following message:

```
   +-----------------------------------------------------------------+
   | TeamConnection Build Processor Started
   | Family: tcpc3
   | Pool: pool1
   | Environment: AIX
   | Build Directory: /home/tcpc3/
   +-----------------------------------------------------------------+
98/11/05 13:23:44 Build Processor Started
98/11/05 13:23:44 Family: tcpc3
98/11/05 13:23:44 Pool: pool1
98/11/05 13:23:44 Environment: AIX
98/11/05 13:23:44 Build Directory: /home/buildtc/tcpc3/
```

**Note:** Notice that this build server will be running in the foreground, and thus, to stop the build server you need to do "Ctrl-C". Also, you cannot do anything else in this window while the build server is running.

3. Later on, when the build server performs build activities, there will be several messages shown in this window, such as:

```
98/11/05 13:41:24 --- Start of Job -----------------------------------
98/11/05 13:41:24 Version: tcpc3|internet1.1|4
98/11/05 13:41:24 Builder: C-compiler.ksh
98/11/05 13:41:24 Output Files: archivo.o
98/11/05 13:41:24 Input Files: archivo.c
98/11/05 13:41:24 Dep. Files: archivo.h
98/11/05 13:41:24 Command: C-compiler.ksh $TC_INPUT $TC_OUTPUT
98/11/05 13:41:25 RC=0    (Expected: == 0) - Success
98/11/05 13:41:25 Storing the build results + looking for more work
98/11/05 13:41:29 --- End of Job -------------------------------------
```

   Note that the above messages are not shown when you start the build server. These messages are just for illustration purposes.

4. Open another window and start the VA TC family server and the notification daemon:

```
teamcd -n mailexit tcpc3 2
```

   Although you could start also the build server with the teamcd command, for clarification purposes, the build server will be started separately.

5. From the window where the family server was started, invoke the GUI for the TeamConnection client:

```
teamcgui &
```

   It is recommended that you add the & symbol at the end in order to invoke the GUI as a background process; in that way you can continue using the window.

6. Create a directory $HOME/work where the files that contain the build scripts involved in the build activities will be located:

```
cd $HOME
mkdir work
```

7. Create a directory $HOME/work/builders where the files related to the builders will be located:

   ```
   mkdir work/builders
   ```

   For more information on why this directory is needed, see “Builders are not versioned: hints on how to keep track of them” on page 30.

8. Update the Settings notebook: From the "TeamConnection - Tasks" window, select the Windows pull down menu and select Settings.

   - Update the Pool field in the Pool page:

     a. Select the right-most tab which is called "Pool".
     b. In the Pool field enter the name used with the -p parameter during the start of the build server, in this case "pool1".
     c. Click on Apply.

   - Update the Relative directory field in the Environment page:

     a. Select the left-most tab which is called "Environment".

     b. In the Relative directory field enter the full path of the top directory where the files will be located: /home/tcpc3/work

     c. Click on Apply.

# CREATION OF SUPPORTING OBJECTS

This chapter describes the creation of the supporting objects (such as components, releases and workareas) that need to be in place in order to handle the build related objects.

## CREATION OF FOUNDATION OBJECTS IN THE FAMILY

As a superuser of the family, create the following foundation objects that will be used in this example:

- A user that will be doing the build activities. It is a good idea to use the superuser id only for those administration activities that require the highest authority.

  By the same token, it is a good idea to use a normal user that is not a superuser in order to demonstrate the necessary authority level that is necessary to perform the build activities.

  In this example, this normal user id will be called "rivera".

  You can use the GUI or the following line command:

  ```
  teamc User  -create  -login rivera -address tcpc3@oem-ppc3 \
              -name "tcpc3" -area buildingTeam -verbose
  ```

- A host list entry for the user "rivera" to login as "tcpc3" from the host "oem-ppc3":

  You can use the GUI or the following line command:

  ```
  teamc Host  -create tcpc3@oem-ppc3 -login rivera -verbose
  ```

- A component "internet" owned by user "rivera" to store the parts for the application.

  The component "root" will be the parent for this component, will use the maintenance process and will accept all the other defaults.

  You can use the GUI or the following line command:

  ```
  teamc Component -create internet -parent root -process maintenance \
                  -owner rivera -description "Internet application"  \
                  -verbose
  ```

- A component "builders" owned by user "rivera" to store the files that contain the information about the builders and the script files for builders.

  The component "internet" will be the parent for this component. The subcomponent will use the maintenance process and will accept all the other defaults.

  You can use the GUI or the following line command:

```
teamc Component -create builders -parent internet -process maintenance \
                -owner rivera -description "Builders repository" -verbose
```

- An authority of "projectlead" in the component "internet" for user "rivera" in order to do all the necessary build and release activities.

  The authority of "projectlead" contains all the actions for "componentlead", "builder", and "releaselead". Thus, the customer may decide to split these roles into several users, instead of concentrating all the roles into one non-superuser as in this example.

  You can use the GUI or the following line command:

  ```
  teamc Access -create  -component internet -login rivera \
               -authority projectlead -verbose
  ```

  Because the component "builders" is a subcomponent of "internet", the subcomponent inherits the authority from the parent component. That is, there is no need to add these authority groups again in the subcomponent.

- A release "internet1.1" owned by user "rivera" to store the parts for the applications.

  The release will be subjected to the authority groups defined in the component "internet", will use the track_driver process and will accept all the other defaults.

  You can use the GUI or the following line command:

  ```
  teamc Release -create internet1.1 -component internet \
                -process track_driver -owner rivera     \
                -description "Internet 1.1" -verbose
  ```

- A release "builders1.1" owned by user "rivera" to store the parts that contain the information about the builders and the script files for builders.

  The release will be subjected to the authority groups inherited by the component "builders", will use the track_driver process and will accept all the other defaults.

  You can use the GUI or the following line command:

  ```
  teamc Release -create builders1.1 -component builders \
                -process track_driver -owner rivera     \
                -description "Builders" -verbose
  ```

## CREATION OF THE WORKAREAS

It is necessary to create workareas in order to create parts in the releases. From now on, the normal user "rivera" will be performing all the actions.

1. Change the TeamConnection userid to "rivera" in the GUI. From the Windows pull down, select Settings and enter "rivera" in the field "Become".

   For the line commands, you can enter:

```
export TC_BECOME=rivera
```

2. Open a feature in the internet component and another feature in the builders component.

   You can use the GUI or the following line commands:

```
teamc Feature -open  -component internet -remarks "Create parts." \
              -prefix f -verbose

teamc Feature -open  -component builders -remarks "Create parts." \
              -prefix f -verbose
```

   In this case, the feature for the internet component is "4" and the one for the builders component is "5".

3. Accept the features.

   You can use the GUI or the following line command:

```
teamc Feature  -accept 4 5 -answer new_function -verbose
```

4. Create a workarea for each accepted feature and its respective release.

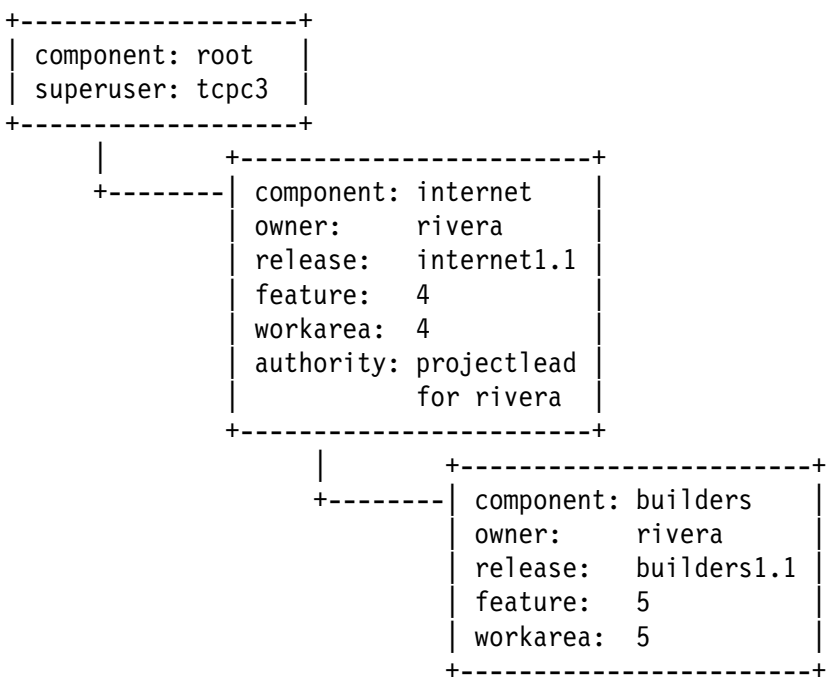
   You can use the GUI or the following line commands:

```
teamc WorkArea -create  -defect 4 -release internet1.1 \
               -owner rivera -verbose

teamc WorkArea  -create  -defect 5 -release builders1.1 \
               -owner rivera -verbose
```

## SUMMARY OF RELEVANT OBJECTS

The following figure shows the relevant objects created so far:


component: root
superuser: tcpc3
component: internet
owner: rivera
release: internet1.1
feature: 4
workarea: 4
authority: projectlead
for rivera
component: builders
owner: rivera
release: builders1.1
feature: 5
workarea: 5

# CREATION OF BUILDERS AND PARSER

This chapter describes the creation of the transformational tools to be used in this example, as well as the builders and parser that are related to them.

## CREATION OF THE TRANSFORMATIONAL TOOLS

This chapter does not show you how to create the builder objects inside Rather, it only shows you how to create the build scripts and store them inside TeamConnection in order to get versioned.

The TeamConnection builders and parsers that will be created in “Create the builders and parsers inside TeamConnection” on page 30 will invoke an actual executable code or shell script (**build scripts**) that needs to be prepared in the workstation.

When the builder is created, then a copy of the build script is stored inside TeamConnection without being versioned. Then when the builder is invoked, a copy of the build script is transferred to the workstation where it is executed.

When a builder is modified and a new executable code is defined, then the previous version of the build script is deleted and the new version is stored.

The sequence shown in this section follows the logical sequence in which these transformational tools are used during a comprehensive build activity (such as the very first build or forcing the build).

**Note:** The builder Null-builder does not use any scripts (it is really a no-op builder). Thus, this section does not have a corresponding transformational tool for this builder.

### C-parser

The executable file $HOME/work/builders/C-parser will be used to parse the C source code and the include header files. It will be compiled locally in AIX based on the sample provided by TeamConnection.

**Note:** In this chapter, for simplicity, only the line commands will be shown; however, the GUI is far easier to use than the line commands.

1. Change the directory to $HOME/work/builders
2. Copy the sample C parser:

```
cp /usr/teamc/samples/c/fhbwpars.c .
```

3. Compile the C parser and name the result "C-parser":

```
xlc fhbwpars.c -o C-parser -D__UNIX__
```

4. Copy the executable C-parser into the $HOME/bin, because it needs to exist in the execution path of the family server at the time a build is performed, and $HOME/bin is part of the PATH in the sample profile:

```
cp C-parser $HOME/bin/.
```

5. Create the parts "C-parser" and its source code "fhbwpars.c" in the release "builders1.1" in order to keep versions for this transformational tool:

```
Create as binary:
teamc Part -create builders/C-parser -release builders1.1 -workarea 5   \
           -component builders -binary  -relative /home/tcpc3/work      \
           -translation no -verbose -remarks "C-parser, AIX exec"

Create as text:
teamc Part -create builders/fhbwpars.c -release builders1.1 -workarea 5 \
           -component builders -text    -relative /home/tcpc3/work      \
           -translation no -verbose -remarks "C-parser, source code"
```

## C-compiler.ksh

The Korn shell script $HOME/work/C-compiler.ksh will be used to invoke the real C compiler (xlc in AIX) to compile the C source code files and the include header files into an object code file. This is called a build script.

Because each builder needs to perform only one single build event, only the compilation function of the real C compiler will be used. No executable code will be linked by this builder. Only the object code will be generated. By the way, the executable code file will be obtained by using the C-linker builder.

To keep this example simple, the intermediary listings from the compilation will not be kept.

There is only one input to the real Compiler: only one source code file.

In short, because this simple builder will have only one input and one output, the input arguments to the Korn shell script will be the following build environment variables:

- $TC_INPUT, for the only input file.
- $TC_OUTPUT, for the only output file.

Furthermore, the following environment variable will be used to pass parameters when performing Part -build events:

- $CFLAGS, for the compiler flags to be passed at compilation time.

Create the following file and mark it as an executable file:

1. Change the directory to where the code will be located:

   ```
   cd $HOME/work/builders
   ```

2. Use an editor to enter the following statements:

```
#!/usr/bin/ksh
#
# Name:  C-compiler.ksh
#
# Purpose: This is a TeamConnection builder that invokes the C compiler

# Process input arguments

source=$1
target=$2

# Process environment variables

flags=$CFLAGS

# Display information (for tracking purposes)

print "C-compiler.ksh: begin"
print "  Processing source file: $source"
print "  Producing object file:  $target"
print "  Using flags:            $flags "

# Use the -c flag to only generate object code and do not invoke
# the linker.
# Use the -o flag to specify the name of the object code file.

xlc -c $source -o $target $flags
rc=$?

# Pass the return code

print "C-compiler.ksh: end; return code=$rc"
exit $rc

# end of file
```

**Figure 6. Korn shell script: C-compiler.ksh**

3. Mark it as an executable file:

   ```
   chmod 755 C-compiler.ksh
   ```

4. Create this build script as a text part into TeamConnection.

   ```
   teamc Part -create  builders/C-compiler.ksh                          \
              -release builders1.1 -workarea 5                          \
              -component builders -text    -relative /home/tcpc3/work    \
              -translation no -verbose -remarks "C-compiler, Korn shell"
   ```

Notice that this is a simple shell script that does not verify if the minimum number of input parameters are passed.  As a learning tool, this shell script displays information that will be very useful later on during the building activities, however, this information might not be needed after the fundamental building concepts are understood and the builder is fully debugged.

## C-linker.ksh

The Korn shell script $HOME/work/C-linker.ksh will be used to invoke the real C compiler (xlc in AIX) to link one or more object code files and to generate an executable code file.

Only the linker function of the real C compiler will be used.  No source code will be processed by this builder.

To keep this example simple, the linking map file will not be kept.

Finally, during the building of the application, we may want to pass to the real compiler some linking flags.

Because this simple builder will have one or more input files, and only one output file, it is NOT convenient to use the passing of input arguments as in the C-compiler.ksh builder. Instead, the Korn shell script will handle the following build environment variables:

- $TC_INPUT, for the one or more input files.
- $TC_OUTPUT, for the only output file.

Furthermore, the following environment variable will be used to pass parameters when performing Part -build events:

- $LFLAGS, for the linker flags.

Create the following file and mark it as an executable file:

1. Change the directory to where the code will be located:

   ```
   cd $HOME/work/builders
   ```

2. Use an editor to enter the following statements:

```
#!/usr/bin/ksh
#
# Name:  C-linker.ksh
#
# Purpose: This is a TeamConnection builder that invokes the C linker

# Instead of processing input arguments, handle the actual build
# environment variables

objects=$TC_INPUT
executable=$TC_OUTPUT
flags=$LFLAGS

# Display information (for tracking purposes)

print "C-linker.ksh: begin"
print "  Processing object files:   $objects"
print "  Producing executable file: $executable"
print "  Using flags:               $flags"

# By NOT using the -c flag, then the linker will be invoked.
# Use the -o flag to specify the name of the executable code file.

xlc $objects -o $executable $flags
rc=$?

# Pass the return code

print "C-linker.ksh: end; return code=$rc"
exit $rc

# end of file
```

**Figure 7. Korn shell script: C-linker.ksh**

3. Mark it as an executable file:

   ```
   chmod 755 C-linker.ksh
   ```

4. Create this build script as a text part into TeamConnection.

   ```
   teamc Part -create  builders/C-linker.ksh                              \
              -release builders1.1 -workarea 5                            \
              -component builders -text    -relative /home/tcpc3/work     \
              -translation no -verbose -remarks "C-linker, Korn shell"
   ```

As a learning tool, this shell script displays information that will be very useful later on during the building activities, however, this information might not be needed after the fundamental building concepts are understood and the builder is fully debugged.

## zipper.ksh

The Korn shell script $HOME/work/zipper.ksh will be used to invoke the Info-zip tool to gather and compress the input files into a single zip file.

Because this simple builder will have one or more input files, and one output file, it is NOT convenient to use the passing of input arguments as in the C-compiler.ksh builder. In fact, it is similar to the C-linker.ksh, in which the Korn shell script will handle the following build environment variables:

- $TC_INPUT, for the one or more input files.
- $TC_OUTPUT, for the only output file.

Furthermore, the following environment variable will be used to pass parameters when performing Part -build events:

- $ZFLAGS, for the zipper flags.

Create the following file and mark it as an executable file:

1. Change the directory to where the code will be located:

   ```
   cd $HOME/work/builders
   ```

2. Use an editor to enter the following statements:

```
#!/usr/bin/ksh
#
# Name:  zipper.ksh
#
# Purpose: This is a TeamConnection builder that invokes Info-zip

# Instead of processing input arguments, handle the actual build
# environment variables

inputs=$TC_INPUT
target=$TC_OUTPUT
flags=$ZFLAGS

# Display information (for tracking purposes)

print "zipper.ksh: begin"
print "  Processing input files: $inputs"
print "  Producing target file:  $target"
print "  Using flags:            $flags"

zip $target $inputs
rc=$?

# Pass the return code

print "zipper.ksh: end; return code=$rc"
exit $rc

# end of file
```

**Figure 8. Korn shell script: zipper.ksh**

3. Mark it as an executable file:

   ```
   chmod 755 zipper.ksh
   ```

4. Create this text part into TeamConnection.

   ```
   teamc Part -create  builders/zipper.ksh                              \
              -release builders1.1 -workarea 5                          \
              -component builders -text    -relative /home/tcpc3/work   \
              -translation no -verbose -remarks "zipper, Korn shell"
   ```

As a learning tool, this shell script displays information that will be very useful later on during the building activities, however, this information might not be needed after the fundamental building concepts are understood and the builder is fully debugged.

## CREATE THE BUILDERS AND PARSERS INSIDE TEAMCONNECTION

After the transformational tools are created in the workstation (see “Creation of the transformational tools” on page 23), then the actual TeamConnection Builders can be created to use these transformational tools.

The creation of a builder in TeamConnection consists of the following steps:

1. If needed, create a script in workstation.

   If the builder requires a script, then this script needs to be created in the client workstation. See “Creation of the transformational tools” on page 23.

   You can also have the build script in the build server by using the file type of "none".

2. Create builder inside TeamConnection.

   A builder, which is a TeamConnection object, needs to be created inside TeamConnection that will invoke a transformational tool by either:

   - Using a script that was copied once from the workstation. In turn, this script will invoke the desired transformational tool. This is the script that was created in the previous step.
   - Invoking directly the transformational tool by using a command input arguments.

**Note:** From now on, to add in the clarification of the concepts, the GUI details will be described, as well as the line commands.

The sequence shown in this section follows the logical sequence in which these transformational tools are used during a comprehensive build activity (such as the very first build or forcing the build).

### Builders are not versioned: hints on how to keep track of them

The builders and parsers stored inside TeamConnection are NOT versioned.

Thus, it might be possible to generate a part with a version of the builder. Later on the builder is changed radically and if a problem is found during the building activities, then unless you made a backup copy of the original builder, you may not recover easily this original version of the builder.

To overcome this problem, in this technical report the source code for the build scripts and the actual description of the builders (obtained from the Information window, by using File -> Save as) are stored inside TeamConnection, under a release (builders1.1) and a component (builders) specifically created for that purpose.

To aid in the recovery of previous versions of builders, or in the migration of builders from one family to another, the information for each builder is extracted into a file, and this file is then created into TeamConnection in the dedicated release and component. This information can be used to create builders with the desired version.

## Definition of the parser C-parser

1. From the Tasks window, select Objects -> Parsers.
2. From the Parsers Filter, select "Show all Parsers" from the History box and click on OK.

   The resulting Parsers window will appear and because it us the first time that we are going to create parsers, it should be empty.
3. Create the parser C-parser by selecting Selected -> Create...
4. From the Create Parser enter:

   ```
   Parser:              C-parser
   Release:             internet1.1
   Command:             C-parser
   Include:             .
   ```

   **Notes:**

   a. Notice that the C-parser executable (specified in the "command" field) needs to exist in the execution path of the family server at the time a build is performed.

      In contrast with the build scripts, the parser code is NOT stored inside TeamConnection.

   b. The parameter "include" can be used to specify other directories beside the current one where to find the include header files. In this case these header files will be located in the same directory as the source file; thus, we need to specify the current directory in the include path.
5. The corresponding line command is:

   ```
   teamc Parser -create C-parser -release internet1.1                \
                -command C-parser -verbose
   ```
6. After the parser is created, refresh (F5) the window, and select the recently created parser and then Selected -> View. The result should be something like this:

   ```
   name               C-parser
   releaseName        internet1.1
   addDate            1998/11/01 05:40:25
   dropDate
   lastUpdate         1998/11/01 05:40:25
   commandName        C-parser
   includePaths
   ```

7. Because the parsers are not versioned and because in the future it might be important to know the values of specific versions, save the output from the "TeamConnection - Information Window" into $HOME/work/builders/C-parsers.info.

8. Create the text part C-parser.info inside the builders1.1 release:

```
teamc Part -create  builders/C-parser.info                          \
           -release builders1.1 -workarea 5                         \
           -component builders -text    -relative /home/tcpc3/work    \
           -translation no -verbose -remarks "C-parser, info"
```

## Definition of the builder C-compiler

1. From the Tasks window, select Objects -> Builders.

2. From the Builders Filter, select "Show all Builders" from the History box and click on OK.

   The resulting Builders window will appear and because it us the first time that we are going to create builders, it should be empty.

3. Create the builder C-compiler by selecting Selected -> Create...

4. From the Create Builder enter:

```
Builder:             C-compiler
Release:             internet1.1
Script:              C-compiler.ksh
Environment:         AIX
Comparison operator: ==
RC value:            0
File type:           text
Source file:         $HOME/work/builders/C-compiler.ksh
Parameters:          \$TC_INPUT \$TC_OUTPUT
Timeout:             1440
```

   **Notes:**

   a. The value for the environment (AIX) must match the same value used with the -e flag when the build server was started.

   b. The fields "comparison operator" and "RC value" indicate (in this case) that the build event should be considered to be successful, only if the return code is equal to 0.

   c. The source file (which is a text file as indicated by "File type") is read from the workstation (indicated by the field "Source file") and it is stored inside TeamConnection.

      After that, the location of this file in the workstation is of no importance to the Builder, and that is the reason why when viewing the information of a Builder, the source file location is not shown. It is very likely to the storage of normal parts in TeamConnection.

d. Notice that because the transformational tool C-compiler.ksh requires two input parameters, the Builder C-compiler passes two parameters to the script.

- During the creation of the builder, in UNIX, you MUST specify the back-slash (\) before the $ (dollar) for TC_INPUT and TC_OUTPUT.
- Notice that during the View action (see next steps), these back-slashes are not shown and unfortunately this may cause confusion.
- If you fail to enter these back-slashes then the proper substitution will not take place during the build time and your builders will fail to perform as expected and the debugging could be very tedious.

e. The C-compiler.ksh script uses the $CFLAGS environment variable, and because it is read as an input argument to the script, it is not necessary to specify $CFLAGS as a builder parameter.

For more information on how to pass build parameters, see “Passing parameters when doing "teamc Part -build"” on page 69.

f. The field "Timeout" specifies the amount of time that the build processor waits for a build script to complete before assuming a failure has occurred. The default is 1,440 minutes (24 hours).

g. If you extract a Builder, the script is extracted to $HOME and it is not extracted it into the "Relative directory" specified in the Settings.

5. The corresponding line command is:

```
teamc Builder -create C-compiler -release internet1.1                 \
              -script C-compiler.ksh -environment AIX -condition "==" \
              -value 0 -from /home/tcpc3/work/builders/C-compiler.ksh \
              -parameters "\$TC_INPUT \$TC_OUTPUT"                    \
              -timeout 1440 -text -verbose
```

6. After the builder is created, refresh (F5) the window, and select the recently created builder and then Selected -> View. The result should be something like this:

```
name              C-compiler
releaseName       internet1.1
parameters        $TC_INPUT $TC_OUTPUT
rebuildTime       1998/10/27 09:39:13
addDate           1998/10/27 09:39:13
dropDate
lastUpdate        1998/10/27 09:39:13
scriptName        C-compiler.ksh
RCCondition       ==
bulkType          text
rcValue           0
environment       AIX
timeout           1440
```

7. Because the builders are not versioned and because in the future it might be important to know the values of specific versions, save the output from the "TeamConnection - Information Window" into $HOME/work/builders/C-compiler.info.

8. Create the text part C-compiler.info inside the builders1.1 release:

```
teamc Part -create  builders/C-compiler.info                          \
           -release builders1.1 -workarea 5                           \
           -component builders -text    -relative /home/tcpc3/work    \
           -translation no -verbose -remarks "C-compiler, info"
```

## Definition of the builder C-linker

1. Create the builder C-linker by selecting Selected -> Create...

2. From the Create Builder enter:

```
Builder:             C-linker
Release:             internet1.1
Script:              C-linker.ksh
Environment:         AIX
Comparison operator: ==
RC value:            0
File type:           text
Source file:         $HOME/work/builders/C-linker.ksh
Parameters:
Timeout:             1440
```

3. The corresponding line command is:

```
teamc Builder -create C-linker   -release internet1.1                 \
              -script C-linker.ksh   -environment AIX -condition "==" \
              -value 0 -from /home/tcpc3/work/builders/C-linker.ksh   \
                                                  -timeout 1440       \
              -text -verbose
```

4. After the builder is created, refresh (F5) the window, and select the recently created builder and then Selected -> View.  The result should be something like this:

```
name               C-linker
releaseName        internet1.1
parameters
rebuildTime        1998/10/27 09:55:34
addDate            1998/10/27 09:55:34
dropDate
lastUpdate         1998/10/27 09:55:34
scriptName         C-linker.ksh
RCCondition        ==
bulkType           text
rcValue            0
environment        AIX
timeout            1440
```

5. Because the builders are not versioned and because in the future it might be important to know the values of specific versions, save the output from the "TeamConnection - Information Window" into $HOME/work/builders/C-linker.info.

6. Create the text part C-linker.info inside the builders1.1 release:

```
teamc Part -create  builders/C-linker.info                          \
           -release builders1.1 -workarea 5                         \
           -component builders -text    -relative /home/tcpc3/work    \
           -translation no -verbose -remarks "C-linker, info"
```

**Notes:**

1. Notice that the script for C-linker.ksh does not read any input arguments, instead, it gets the values from the $TC_INPUT and the $TC_OUTPUT environment variables that are prepared by the Builder at build time.

2. The C-linker.ksh script uses the $LFLAGS environment variable, and because it is read as an input argument to the script, it is not necessary to specify $LFLAGS as a builder parameter.

   For more information on how to pass build parameters, see “Passing parameters when doing "teamc Part -build"” on page 69.

3. A linker may have one or more input object code files. That is the reason for not passing the list as an input argument, but by using the environment variable $TC_INPUT.

## Definition of the builder zipper

1. Create the builder zipper by selecting Selected -> Create...

2. From the Create Builder enter:

```
Builder:             zipper
Release:             internet1.1
Script:              zipper.ksh
Environment:         AIX
Comparison operator: ==
RC value:            0
File type:           text
Source file:         $HOME/work/builders/zipper.ksh
Parameters:
Timeout:             1440
```

3. The corresponding line command is:

```
teamc Builder -create zipper   -release internet1.1                 \
              -script zipper.ksh     -environment AIX -condition "==" \
              -value 0 -from /home/tcpc3/work/builders/zipper.ksh     \
                                                 -timeout 1440      \
              -text -verbose
```

4. After the builder is created, refresh (F5) the window, and select the recently created builder and then Selected -> View. The result should be something like this:

```
name                zipper
releaseName         internet1.1
parameters
rebuildTime         1998/10/27 10:46:53
addDate             1998/10/27 10:46:53
dropDate
lastUpdate          1998/10/27 10:46:53
scriptName          zipper.ksh
RCCondition         ==
bulkType            text
rcValue             0
environment         AIX
timeout             1440
```

5. Because the builders are not versioned and because in the future it might be important to know the values of specific versions, save the output from the "TeamConnection - Information Window" into $HOME/work/builders/zipper.info.

6. Create the text part zipper.info inside the builders1.1 release:

```
teamc Part -create  builders/zipper.info                              \
           -release builders1.1 -workarea 5                           \
           -component builders -text    -relative /home/tcpc3/work    \
           -translation no -verbose -remarks "zipper, info"
```

**Notes:**

1. Notice that the script for zipper.ksh does not read any input arguments, instead, it gets the values from the $TC_INPUT and the $TC_OUTPUT environment variables that are prepared by the Builder at build time.

2. The zipper.ksh script uses the $ZFLAGS environment variable, and because it is read as an input argument to the script, it is not necessary to specify $ZFLAGS as a builder parameter.

   For more information on how to pass build parameters, see “Passing parameters when doing "teamc Part -build"” on page 69.

3. A zipper may have one or more input object code files. That is the reason for not passing the list as an input argument, but by using the environment variable $TC_INPUT.

## Definition of the builder Null-builder

1. Create the builder Null-builder by selecting Selected -> Create...

2. From the Create Builder enter:

```
Builder:             Null-builder
Release:             internet1.1
Script:              null
Environment:         AIX
Comparison operator: ==
RC value:            0
File type:           none
Source file:
Parameters:
Timeout:             1440
```

3. The corresponding line command is:

```
teamc Builder -create Null-builder -release internet1.1               \
              -script null           -environment AIX -condition "==" \
              -value 0                                                \
                                                 -timeout 1440        \
              -none -verbose
```

4. After the builder is created, refresh (F5) the window, and select the recently created builder and then Selected -> View. The result should be something like this:

```
name               Null-builder
releaseName        internet1.1
parameters
rebuildTime        1998/10/27 10:05:20
addDate            1998/10/27 10:05:20
dropDate
lastUpdate         1998/10/27 10:05:20
scriptName         null
RCCondition        ==
bulkType           none
rcValue            0
environment        AIX
timeout            1440
```

5. Because the builders are not versioned and because in the future it might be important to know the values of specific versions, save the output from the "TeamConnection - Information Window" into $HOME/work/builders/Null-builder.info.

6. Create the text part Null-builder.info inside the builders1.1 release:

```
teamc Part -create  builders/Null-builder.info                        \
           -release builders1.1 -workarea 5                           \
           -component builders -text    -relative /home/tcpc3/work    \
           -translation no -verbose -remarks "Null-builder, info"
```

**Notes:**

1. Do not use the name "null" for the Builder Name because "null" is a reserved word by TC.

   To modify a Builder object in order to describe that no builder tools should be use, specify in the script name the keyword "null".

2. Notice that the Null-builder has a File type of none and has no source file and no parameters.

# CREATION OF PARTS AND BUILD TREE

This chapter describes the creation of the parts and their relationships (which form the build tree) involved in the build activities.

## CREATE THE FILES IN THE WORKSTATION

The source code file, the include header file and the text files need to exist in the workstation in order to create the corresponding part contents inside the TeamConnection family.

### Create the include header file "archivo.h"

Create the include header file "archivo.h".

1. Change the directory to where the code will be located:
   ```
   cd $HOME/work
   ```
2. Use an editor to enter the following statements:

```
/*
NAME:    archivo.h
*/

#define  MY_DEFINE 1

/* end of file */
```

**Figure 9. Source code for include header file "archivo.h"**

### Create the source code file "archivo.c"

Create the following source code file:

1. Change the directory to where the code will be located:
   ```
   cd $HOME/work
   ```
2. Use an editor to enter the following statements:

```
/*
NAME:    archivo.c
PURPOSE: C source code for a sample that shows the Build function
         in TeamConnection.
         It includes a single header file.
         It does not require input arguments.
*/

#include <stdio.h>
#include "archivo.h"

main(int argc, char **argv)
{
   printf("Hola! This is the output of running 'archivo'.\n");
   return 0;
}

/* end of file */
```

**Figure 10. Source code for file "archivo.c"**

## Create the end-user notes: text file "readme.txt"

Create the following readme file that contains end-user information about the sample application:

1. Change the directory to where the code will be located:

   ```
   cd $HOME/work
   ```

2. Use an editor to enter the following statements:

```
Subject: These are the release notes for the application 'archivo'

This is the first release.
To run the application enter:  archivo

*** end of file ***
```

**Figure 11. Text for the readme file "readme.txt"**

### Create the developer's notes: text file "devnotes.txt"

Create the following readme file that contains developer's information about the sample application:

1. Change the directory to where the code will be located:

   ```
   cd $HOME/work
   ```

2. Use an editor to enter the following statements:

```
Subject: These are the developer's notes for the application 'archivo'

To build for debugging, specify the build parameter CFLAGS=-g.

*** end of file ***
```

**Figure 12. Text for the readme file "devnotes.txt"**

## CREATE THE PARTS: IF NEEDED, SPECIFY ONLY THE BUILDER

Now that the builders and the parser are ready in TeamConnection, and the source files exist in the workstation, the next step is to create the parts, whether these parts exist in the workstation (such as the source code) or these parts will be generated later by a builder (such as the object code) and for the moment they are created as **empty parts** ("place-holders"), that is, with no immediate contents.

In the case of an output part (for example the object code file), during the build event the build script will be extracted as well as the input parts (for example, the source code files), then these input parts will be processed to generate the output part and this output part will be checked in inside the family.

At this stage, for novice users we would recommend to not define yet the build relationships between the parts, that is, when creating a part, do not specify which part is the parent (if any). This will be done in a separate step, in “Connecting the parts to form the build tree” on page 51.

The idea is to first create the parts, and later on define their build relationships. Once you are familiar with the Build function, you may combine these 2 steps (create the part AND specify the relationship with another part).

However, at this stage it is important to define, if needed, the builder associated with this part.

**Key Question:** When creating parts, the key question to be asked in order to determine if there is a builder associated with this part is:

```
What is the tool that generates this part?
```

For example, when creating the source code file archivo.c the answer is "none" because there is no automated builder. When asking this question for the object file archivo.o, the answer is "archivo.o is the output of the C-compiler", and thus, the builder for archivo.o is C-compiler.

To create the parts for this example, do the following:

1. From the Tasks Window, select Objects -> Parts -> BuildView... and from the BuildView Filter specify all the files (by selecting baseName, with the SQL "like" and the wildcard "%") for the desired release, such as "internet1.1" and the desired workarea, such as "4":

   ```
   Release: internet1.1       Workarea:  4
   baseName like '%'
   ```

   The corresponding line commands are:

   ```
   teamc report -raw -view PartView -where "baseName like '%'"   \
                -release internet1.1  -workarea 4
   ```

   Do not select Parts... or PartsFull... because these windows, although are related to parts, will not show the options used by the Build function.

Create the parts in the following subsections. When appropriate specify the Builder, but DO NOT specify the Parent information. This will be done later on.

Only one example of the line command to create a part will be shown in this section: the one for the first part in the list. You can use it as the basis for creating the other parts.

If you are using the GUI, it is recommended that you use the Apply button instead of the OK button, in that way you just need to enter the information that is different, such as the part name.

When using BuildView, notice the following about the parts shown in this window:

- The parts that do not have a builder have an icon of a blank sheet of paper with the upper right corner folded.
- The parts that have a builder have an icon that has 3 pieces:
  - The left most represents several files.
  - The middle is the arrow that shows that there is a transformation.
  - The right most represents the output file (this file that is associated with the builder).
- The icon for the parts that have a builder is black to represent that the part is out of date.

When the part is successfully built (that is, it is up to date), the color of the icon is green.

**Note:** The creation of the parts needs to be done from the workstation that has the actual source files. Once the parts are created inside TeamConnection, the rest of the build activities can be performed from another TeamConnection client if you wish it (provided that you use the same Pool value in the Pool page of the Settings notebook).

## Create the include header part "archivo.h"

Create the include header part as text part into TeamConnection:

From the GUI:

```
Part name:          archivo.h
Release:            internet1.1
Work area:          4
Component:          internet
File type:          Text
Source:             same
Source file:
Source directory:   $HOME/work
Remarks:            Include header file
Parent:
Parent type:
Relation to parent:
Builder:
Parameters:
Parser:             C-parser
Temporary file:
Translation:        no
```

The line command is:

```
teamc Part -create  archivo.h                                          \
           -release internet1.1  -workarea 4                           \
           -component internet -text    -relative /home/tcpc3/work     \
           -translation no -verbose -remarks "Include header file"     \
           -parser C-parser
```

The following informational messages are displayed when everything went fine:

```
6021-301 Invoking Parser C-parser for archivo.h
6021-303 A successful parse resulted from using the parser C-parser. The
         parser return code is 0
Part archivo.h was created successfully.
```

### What to do when the parsing is not successful

The following list contains some possible error messages during the parsing and the way to overcome them:

- If the C-parser executable is not in the PATH of the TeamConnection family server user id, such as the directory $HOME/bin, then you will get the following error message:

  ```
  6021-301 Invoking Parser C-parser for archivo.h
  6021-302 An unsuccessful parse resulted from using the parser C-parser for
           part archivo.h. The parser return code is 32512.
  ```

- If LIBPATH does not have /usr/lib (or other necessary libraries) in the TeamConnection family server user id, then you will get the following error message:

  ```
  6021-301 Invoking Parser C-parser for archivo.h
  6021-302 An unsuccessful parse resulted from using the parser C-parser for
           part archivo.h. The parser return code is -256.
  ```

  In this sample application, the LIBPATH had to be:

  ```
  export LIBPATH=/usr/lib:$TC_HOME/lib
  ```

- Some other debugging ideas are:
  - Replace temporarily the file $HOME/bin/C-parser with the following Korn shell script that will list the input arguments that are passed by the TeamConnection family to the parser:

```
#!/usr/bin/ksh
#
# Name:  debug.parser.ksh
#
# Purpose: This is a debug parser that shows the input arguments

print "debug.parser.ksh: begin"    > $HOME/parser.out
print "input parameters: $*"      >> $HOME/parser.out
rc=0
print "debug.parser.ksh: end; return code=$rc" >> $HOME/parser.out
exit $rc

# end of file
```

**Figure 13. Korn shell script: debug.parser.ksh**

The output in $HOME/parser.out will be something like this:

```
debug.parser.ksh: begin
input parameters: /tmp/jaa1234 /tmp/kaa1234 tcpc3 4 internet1.1
debug.parser.ksh: end; return code=0
```

Where the input arguments are:

- input file (file to be parsed)
- file with the list of dependencies, one entry per dependency.
- family name
- workarea name
- release name

Note that the input file and the file with the list of dependencies are placed as temporary files in /tmp. These temporary files are deleted by the TeamConnection family server after the parsing.

– Run the parser with the proper input arguments, such as:

```
C-parser archivo.h file.out tcpc3 4 internet1.1
```

The contents in file.out is empty for archivo.h. But for archivo.c, it will have the following 2 entries:

```
stdio.h
archivo.h
```

## Create the source code part "archivo.c"

Create the source code part as text part into TeamConnection:

From the GUI:

```
Part name:          archivo.c
Release:            internet1.1
Work area:          4
Component:          internet
File type:          Text
Source:             same
Source file:
Source directory:   $HOME/work
Remarks:            Source code file
Parent:
Parent type:
Relation to parent:
Builder:
Parameters:
Parser:             C-parser
Temporary file:
Translation:        no
```

The line command is:

```
teamc Part -create  archivo.c                                          \
           -release internet1.1  -workarea 4                           \
           -component internet -text    -relative /home/tcpc3/work     \
           -translation no -verbose -remarks "Source code file"        \
           -parser C-parser
```

**Notes:**

1. The parser will identify the following two include dependencies:

   ```
   stdio.h
   archivo.h
   ```

2. Notice that in the BuildView window, this is the only part so far that has a '+' sign in the left side, which indicates that there are relationships with other parts. In this case, the only build relationship is the one identified by the parser.

3. The expansion of the build tree in the Unix GUI is a bit tricky:

   a. Click on the '+' once. It seems that this retrieves additional data.
   b. Click again on the '+' and now you should see the expansion.

4. When you click on the '+' sign, the following TeamConnection command is actually invoked:

   ```
   tcbv archivo.col -type TCPart -rec -rel internet1.1 -work 4
   ```

## Create the object code part "archivo.o"

Create the object code part "archivo.o".

This part does not have a real counterpart in the workstation (of course, only if you extract the part to the workstation).

This part is initially an empty part that will be used to store inside TeamConnection the binary output of the builder. Thus, notice that the "Source" field is "no source" (-empty).

From the GUI:

```
Part name:          archivo.o
Release:            internet1.1
Work area:          4
Component:          internet
File type:          Binary
Source:             no source
Source file:
Source directory:
Remarks:            This is object code.
Parent:
```

```
Parent type:
Relation to parent:
Builder:            C-compiler
Parameters:
Parser:
Temporary file:
Translation:        no
```

The line command to create the above part is:

```
teamc Part -create  archivo.o
           -release internet1.1  -workarea 4                          \
           -component internet -binary -empty -builder C-compiler     \
           -translation no -verbose                                   \
           -remarks "This is object code."
```

**Note:** A TeamConnection part is an object, whether it has contents or not. Thus, a part that is created with no contents (empty) is still a part.

## Create the executable code part "archivo"

Create the executable code part "archivo".

This part does not have a real counterpart in the workstation (of course, only if you extract the part to the workstation).

This part is initially an empty part that will be used to store inside TeamConnection the binary output of the builder. Thus, notice that the "Source" field is "no source" (-empty).

From the GUI:

```
Part name:          archivo
Release:            internet1.1
Work area:          4
Component:          internet
File type:          Binary
Source:             no source
Source file:
Source directory:
Remarks:            This is executable code.
Parent:
Parent type:
Relation to parent:
Builder:            C-linker
Parameters:
Parser:
Temporary file:
Translation:        no
```

The line command to create the above part is:

```
teamc Part -create  archivo
           -release internet1.1  -workarea 4                          \
           -component internet -binary -empty -builder C-linker       \
           -translation no -verbose                                    \
           -remarks "This is executable code."
```

## Create the text part "readme.txt"

Create the text part that contains the release notes, "readme.txt":

From the GUI:

```
   Part name:          readme.txt
   Release:            internet1.1
   Work area:          4
   Component:          internet
   File type:          Text
   Source:             same
   Source file:
   Source directory:   $HOME/work
   Remarks:            Release notes, readme file
   Parent:
   Parent type:
   Relation to parent:
   Builder:
   Parameters:
   Parser:
   Temporary file:
   Translation:        no
```

The line command to create the above part is:

```
teamc Part -create  readme.txt                                         \
           -release internet1.1  -workarea 4                           \
           -component internet -text    -relative /home/tcpc3/work     \
           -translation no -verbose                                    \
           -remarks "Release notes, readme file"
```

## Create the zipper part "archivo.zip"

Create the zipper part "archivo.zip".

This part is initially an empty part that will be used to store inside TeamConnection the binary output of the builder.  Thus, notice that the "Source" field is "no source" (-empty).

From the GUI:

```
Part name:          archivo.zip
Release:            internet1.1
Work area:          4
Component:          internet
File type:          Binary
Source:             no source
Source file:
Source directory:
Remarks:            This is a zip file.
Parent:
Parent type:
Relation to parent:
Builder:            zipper
Parameters:
Parser:
Temporary file:
Translation:        no
```

The line command to create the above part is:

```
teamc Part -create  archivo.zip
           -release internet1.1  -workarea 4                          \
           -component internet -binary -empty -builder zipper         \
           -translation no -verbose                                   \
           -remarks "This is a zip file."
```

## Create the text part "devnotes.txt"

Create the text part that contains the developer's notes, "devnotes.txt":

From the GUI:

```
Part name:          devnotes.txt
Release:            internet1.1
Work area:          4
Component:          internet
File type:          Text
Source:             same
Source file:
Source directory:   $HOME/work
Remarks:            Developer's notes, readme file
Parent:
Parent type:
Relation to parent:
Builder:
Parameters:
Parser:
Temporary file:
Translation:         no
```

The line command to create the above part is:

```
teamc Part -create  devnotes.txt                                    \
           -release internet1.1  -workarea 4                        \
           -component internet -text    -relative /home/tcpc3/work  \
           -translation no -verbose                                 \
           -remarks "Developer's notes, readme file"
```

## Create the collector part "archivo.col"

Create the collector part "archivo.zip".

This part is not associated at all with a file in the workstation, that is, there is nothing to extract.

This is just a part that serves as a collector of other parts.  Notice that the "File type" is "None" (-none).

From the GUI:

```
   Part name:          archivo.col
   Release:            internet1.1
   Work area:          4
   Component:          internet
   File type:          None
   Source:
   Source file:
   Source directory:
   Remarks:            This is a collector part.
   Parent:
   Parent type:
   Relation to parent:
   Builder:            Null-builder
   Parameters:
   Parser:
   Temporary file:
   Translation:        no
```

The line command to create the above part is:

```
teamc Part -create  archivo.col
           -release internet1.1  -workarea 4                        \
           -component internet -none          -builder Null-builder \
           -translation no -verbose                                 \
           -remarks "This is a collector part."
```

## CONNECTING THE PARTS TO FORM THE BUILD TREE

Once the parts are created (see “Create the parts: if needed, specify only the builder” on page 41), you can now proceed to form the build tree by connecting the parts.

**Key Question:** When creating the build tree, the key question to be asked when connecting the selected part is:

```
Which part is the output of the builder for which this selected part
is an input?
In other words, what is the parent part for this selected part?
```

Start from the bottom of the build tree, that is, start with input part or child part. This is the best strategy because the questions from the dialog box are geared towards it and the default relationship between the child and the parent is "input", that is, the child is the input to the builder that generates the parent (output).

**Notes:**

1. You need to specify only the build relationships.

   The parser relationships are specified by the parser script as attributes of input parts. Thus, do not try to specify any parser relationships.

2. If you decide to start at the top of the build tree, then you need to think in terms of a "negative logic"; because it may be very confusing for novice users this approach is not recommended.

3. The documentation does not explain in a clear manner what is the role of parentType, which is by default "TCPart". Do not use anything else (unless you are using a tool that exploits the repository API from VA TC and new part types are defined).

   If you use another value, you will get the confusing message:

   ```
   6021-103 The latest version of Part z was not found for work area x
   and release y.
   ```

4. In all commands, the flag "-type TCPart" can be omitted because it is the default.

   This flag is used by the GUI and thus, it is shown in the teamcv3.log file from where we got copied the actual line commands used by the GUI.

5. Once the parts are created inside TeamConnection, the rest of the build activities can be performed from another TeamConnection client if you wish it (provided that you use the same Pool value in the Pool page of the Settings notebook).

## Connect the source code "archivo.c" to the object code "archivo.o"

Connect the source code "archivo.c" to the object code "archivo.o".

1. Select the part archivo.c, then Selected -> Connect...(>)
2. Fill out the dialog box:

```
Path name:       archivo.c
Type:            TCPart
Release:         internet1.1
Work area:       4
Parent:          archivo.o
Parent type:
Relation to parent: input
```

3. Click on Apply.  In that way you can reuse the window for the other files.
4. The corresponding line command is:

```
teamc Part -connect archivo.c  -parent archivo.o -input              \
           -type TCPart -release internet1.1  -workarea 4 -verbose
```

## Connect the object code "archivo.o" to the executable code "archivo"

Connect the object code "archivo.o" to the executable code "archivo".

1. Select the part archivo.o, then Selected -> Connect...(>)
2. Fill out the dialog box:

```
Path name:       archivo.o
Type:            TCPart
Release:         internet1.1
Work area:       4
Parent:          archivo
Parent type:
Relation to parent: input
```

3. Click on Apply.  In that way you can reuse the window for the other files.
4. The corresponding line command is:

```
teamc Part -connect archivo.o  -parent archivo   -input              \
           -type TCPart -release internet1.1  -workarea 4 -verbose
```

## Connect the executable code "archivo" to the zip part "archivo.zip"

Connect the executable code "archivo" to the zip part "archivo.zip".

1. Select the part archivo, then Selected -> Connect...(>)
2. Fill out the dialog box:

```
Path name:       archivo
Type:            TCPart
Release:         internet1.1
Work area:       4
Parent:          archivo.zip
Parent type:
Relation to parent: input
```

3. Click on Apply.  In that way you can reuse the window for the other files.
4. The corresponding line command is:

```
teamc Part -connect archivo    -parent archivo.zip -input             \
           -type TCPart -release internet1.1  -workarea 4 -verbose
```

## Connect the release notes "readme.txt" to the zip part "archivo.zip"

Connect the release notes "readme.txt" to the zip part "archivo.zip".

1. Select the part readme.txt, then Selected -> Connect...(>)
2. Fill out the dialog box:

```
Path name:       readme.txt
Type:            TCPart
Release:         internet1.1
Work area:       4
Parent:          archivo.zip
Parent type:
Relation to parent: input
```

3. Click on Apply.  In that way you can reuse the window for the other files.
4. The corresponding line command is:

```
teamc Part -connect readme.txt -parent archivo.zip -input             \
           -type TCPart -release internet1.1  -workarea 4 -verbose
```

## Connect the zip "archivo.zip" to the collector part "archivo.col"

Connect the zip "archivo.zip" to the collector part "archivo.col".

1. Select the part archivo.zip, then Selected -> Connect...(>)
2. Fill out the dialog box:

```
Path name:       archivo.zip
Type:            TCPart
Release:         internet1.1
Work area:       4
Parent:          archivo.col
Parent type:
Relation to parent: input
```

3. Click on Apply.  In that way you can reuse the window for the other files.
4. The corresponding line command is:

```
teamc Part -connect archivo.zip -parent archivo.col -input           \
           -type TCPart -release internet1.1  -workarea 4 -verbose
```

## Connect the developer's notes "devnotes.txt" to the zip part "archivo.zip"

Connect the developer's notes "devnotes.txt" to the zip part "archivo.zip".

1. Select the part devnotes.txt, then Selected -> Connect...(>)
2. Fill out the dialog box:

```
Path name:       devnotes.txt
Type:            TCPart
Release:         internet1.1
Work area:       4
Parent:          archivo.col
Parent type:
Relation to parent: input
```

3. Click on OK.
4. The corresponding line command is:

```
teamc Part -connect devnotes.txt -parent archivo.col -input          \
           -type TCPart -release internet1.1  -workarea 4 -verbose
```

## How to interpret the output of "View Information" on build objects

Once the parts and their build relationships have been created, you can see the appropriate information related to a part in TeamConnection by selecting the part and then choose Selected -> View -> View information ...

- The best way to interpret the information in the field "typeOfRelation" of the section "build relationships" is as follows:
  - DependsOn

    This is a parsing relationship which usually indicates that the part that is described in detail in the information window is an include header part to another part, such as a source code or another include header part.

    Interpret it as "archivo.c depends on (or in a sense, it is an "output" of) the include header file archivo.h". That is, if the include header archivo.h is changed, then the source code archivo.c is marked as out-of-date.

    This means that during subsequent checkins of archivo.h, all the parts that are shown with a relationship of "DependsOn" will be marked as needed to be rebuilt. That is the reason that in complex build trees, a checkin action of an include header part that is needed by many parts, will take a long time to complete.
  - DependentOf

    This is a parsing relationship which usually indicates that the part that is described in detail in the information window is a source part that includes other parts, such as an include header part.

    Interpret it as "archivo.h is a dependent of (or in a sense, it is an input to) the source code archivo.c". That is, if the source code archivo.c is changed, then the include header archivo.h is not affected at all.
  - InputTo

    Interpret it as "archivo.o is InputTo the builder (C-linker) that generates archivo"
  - OutputOf

    Interpret it as "archivo.o is OutputOf the builder (C-compiler) in which archivo.c is an input file"
- The field "buildStatus" can have the following values:
  - success

    This means that either the part is up to date, or that the part is not an output of any builder (such as readme.txt or archivo.h).
  - out_of_date

This means that the part is an output of a builder and this part has just been created or that the input parts for the builder have a more recent time stamp.

This is the information for the parts:

- archivo.h

```
buildStatus:    success

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo.c                                 TCPart     DependsOn
```

- archivo.c

```
buildStatus:    success

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo.o                                 TCPart     OutputOf
   archivo.h                                 TCPart     DependentOf
```

- archivo.o

```
buildStatus:    out_of_date

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo                                   TCPart     OutputOf
   archivo.c                                 TCPart     InputTo
```

- archivo

```
buildStatus:    out_of_date

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo.zip                               TCPart     OutputOf
   archivo.o                                 TCPart     InputTo
```

- readme.txt

```
buildStatus:    success

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo.zip                               TCPart     OutputOf
```

- archivo.zip

```
buildStatus:    out_of_date

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo.col                               TCPart     OutputOf
   archivo                                   TCPart     InputTo
   readme.txt                                TCPart     InputTo
```

- devnotes.txt

```
buildStatus:    success

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   archivo.col                               TCPart     OutputOf
```

- archivo.col

```
buildStatus:    out_of_date

build relationships:
   pathName                                  partType   typeOfRelation
   ----------------------------------------- ---------- --------------
   devnotes.txt                              TCPart     InputTo
   archivo.zip                               TCPart     InputTo
```

## ACTUAL BUILD TREE IN THE BUILDVIEW WINDOW

The actual build tree for the parts that were connected in the previous step is shown in “Representation of the build tree in the BuildView window” on page 11.

# PERFORMING BUILD EVENTS

This chapter describes how the Build function handles different kinds of build events.

## PERFORMING BUILD EVENTS THAT AFFECT FEW PARTS

This section shows some build events that involve few parts at a time and do not trigger other build events. This is not the normal mode of using the Build function; however, the idea of this section is to explore in detail how a single build event is done.

### Building the object code "archivo.o"

1. Select archivo.o and specify the action Build.

   The corresponding line command is:

   ```
   teamc Part -build archivo.o    -type TCPart -release internet1.1    \
              -workarea 4 -pool pool1 -normal -verbose
   ```

2. You will see the following messages in the Build Progress window:

   ```
   6021-700 Number of distinct build events for this build: 1.
   Build of 'archivo.o' started at '1998/11/03 10:35:11'
   via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
   Build of 'archivo.o' successfully completed at '1998/11/03 10:35:23'.
   Completed Jobs: 1
   Remaining Jobs: 0
   Processing Completed for 'archivo.o'.
   ```

3. Select archivo.o and specify the action View Build Messages:

   The corresponding line command is:

   ```
   teamc Part -viewmsg archivo.o   -type TCPart -release internet1.1  \
              -workarea 4 -verbose
   ```

4. You will see the following messages in the Information Window:

   ```
   There is not a parser associated with this
   part object.
   ****************** Messages from the builder ******************
   Version: tcpc3|internet1.1|4
   Builder: C-compiler.ksh
   Output Files: archivo.o
   Input Files: archivo.c
   Dep. Files: archivo.h
   Command: C-compiler.ksh $TC_INPUT $TC_OUTPUT
   RC=0    (Expected: == 0) - Success
   ```

```
****************** Build Output Follows ************************
C-compiler.ksh: begin
  Processing source file: archivo.c
  Producing object file:  archivo.o
  Using flags:
C-compiler.ksh: end; return code=0
****************** End Of Build Output *************************
```

5. Now the View Information action shows that the buildStatus for archivo.o changed from:

   ```
   buildStatus      out_of_date
   ```

   To the new value:

   ```
   buildStatus      success
   ```

6. Refresh the window and you will see that the icon for this part now has a green check mark in the smaller triangle on the right side of the icon. This indicates that the status is "success".

### Build again "archivo.o" (nothing to do!)

Just for testing purposes, try to build again the object code "archivo.o", which is now up-to-date. The expected result is that nothing should be done.

1. Select the part archivo.o and specify the Build action.
2. You will see the following message in the Build progress window:

   ```
   Warning: No translations performed.  Everything is up to date.
   ```

3. If you look at the View Build Message you will see the previous message about the compiler, which is actually correct, because there were NO build events at this time and thus, there was no need to replace the build message.

### Building the executable code "archivo"

1. Select the executable part "archivo" and specify the action Build.

   The corresponding line command is:

   ```
   teamc Part -build archivo      -type TCPart -release internet1.1    \
              -workarea 4 -pool pool1 -normal -verbose
   ```

2. You will see the following messages in the Build Progress window:

```
6021-700 Number of distinct build events for this build: 1.
Build of 'archivo' started at '1998/11/03 13:47:41'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo' successfully completed at '1998/11/03 13:47:45'.
Completed Jobs: 1
Remaining Jobs: 0
Processing Completed for 'archivo'.
```

3. Select archivo and specify the action View Build Messages:

   The corresponding line command is:

   ```
   teamc Part -viewmsg archivo     -type TCPart -release internet1.1  \
              -workarea 4 -verbose
   ```

4. You will see the following messages in the Information Window:

   ```
   There is not a parser associated with this
   part object.
   ****************** Messages from the builder ******************
   Version: tcpc3|internet1.1|4
   Builder: C-linker.ksh
   Output Files: archivo
   Input Files: archivo.o
   Command: C-linker.ksh
   RC=0    (Expected: == 0) - Success
   ****************** Build Output Follows ************************
   C-linker.ksh: begin
     Processing object files:   archivo.o
     Producing executable file: archivo
     Using flags:
   C-linker.ksh: end; return code=0
   ****************** End Of Build Output ************************
   ```

5. Now the View Information action shows that the buildStatus for archivo changed from:

   ```
   buildStatus      out_of_date
   ```

   To the new value:

   ```
   buildStatus      success
   ```

6. Refresh the window and you will see that the icon for this part now has a green check mark in the smaller triangle on the right side of the icon.  This indicates that the status is "success".

### Extract the executable "archivo" to verify that the build was fine

To verify that the executable code "archivo" was indeed successfully built, it is necessary to extract it and to run it in an AIX workstation.

1. Select the executable code part "archivo" and specify the action "Extract".

2. The corresponding line command is:

```
teamc Part -extract archivo      -release internet1.1 -workarea 4    \
           -relative /home/tcpc3/work -crlf -verbose
```

3. Execute the file:

```
$  ./archivo
```

4. The output should be:

```
Hola! This is the output of running 'archivo'.
```

## Building the zip part "archivo.zip"

1. Select the zip part "archivo.zip" and specify the action Build.

   The corresponding line command is:

```
teamc Part -build archivo.zip  -type TCPart -release internet1.1    \
           -workarea 4 -pool pool1 -normal -verbose
```

2. You will see the following messages in the Build Progress window:

```
6021-700 Number of distinct build events for this build: 1.
Build of 'archivo.zip' started at '1998/11/04 04:42:17'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo.zip' successfully completed at '1998/11/04 04:42:21'.
Completed Jobs: 1
Remaining Jobs: 0
Processing Completed for 'archivo.zip'.
```

3. Select archivo.zip and specify the action View Build Messages:

   The corresponding line command is:

```
teamc Part -viewmsg archivo.zip -type TCPart -release internet1.1  \
           -workarea 4 -verbose
```

4. You will see the following messages in the Information Window:

```
There is not a parser associated with this
part object.
****************** Messages from the builder ******************
Version: tcpc3|internet1.1|4
Builder: zipper.ksh
Output Files: archivo.zip
Input Files: archivo readme.txt
Command: zipper.ksh
RC=0    (Expected: == 0) - Success
****************** Build Output Follows ************************
zipper.ksh: begin
  Processing input files: archivo readme.txt
  Producing target file:  archivo.zip
  Using flags:
```

```
  adding: archivo (deflated 58%)
  adding: readme.txt (deflated 30%)
zipper.ksh: end; return code=0
****************** End Of Build Output ************************
```

5. Now the View Information action shows that the buildStatus for archivo changed from:

   ```
   buildStatus      out_of_date
   ```

   To the new value:

   ```
   buildStatus      success
   ```

6. Refresh the window and you will see that the icon for this part now has a green check mark in the smaller triangle on the right side of the icon. This indicates that the status is "success".

### Extract the zip "archivo.zip" to verify that the build was fine

To verify that the zip part "archivo.zip" was indeed successfully built, it is necessary to extract it and run "unzip -l" to see its contents.

1. Select the zip part "archivo.zip" and specify the action "Extract".
2. The corresponding line command is:

   ```
   teamc Part -extract archivo.zip  -release internet1.1 -workarea 4    \
              -relative /home/tcpc3/work -crlf -verbose
   ```

3. Run the command:

   ```
   $ unzip -l ./archivo
   ```

4. The output should be:

   ```
   Archive:  archivo.zip
    Length    Date    Time    Name ("[" ==> case
    ------    ----    ----    ----   conversion)
      2885  11-04-98  04:42   archivo
       155  11-04-98  04:42   readme.txt
    ------                    -------
      3040                    2
   ```

### Building the collector part "archivo.col"

1. Select the collector part "archivo.col" and specify the action Build.

   The corresponding line command is:

   ```
   teamc Part -build archivo.col  -type TCPart -release internet1.1    \
              -workarea 4 -pool pool1 -normal -verbose
   ```

2. You will see the following messages in the Build Progress window:

```
6021-700 Number of distinct build events for this build: 1.
Build of 'archivo.col' successfully completed at '1998/11/04 05:25:47'.
Completed Jobs: 1
Remaining Jobs: 0
Processing Completed for 'archivo.col'.
```

3. Select archivo.col and specify the action View Build Messages:

   The corresponding line command is:

   ```
   teamc Part -viewmsg archivo.col -type TCPart -release internet1.1  \
              -workarea 4 -verbose
   ```

4. You will see the following messages in the Information Window:

   ```
   There is not a parser associated with this
   part object.
   There were no builder messages associated with this
   part object using builder Null-builder
   ```

5. Bug in code (hopefully to be fixed after fixpak 3.0.1).

   Due to a bug in the code with the handling of Null Builders in an standalone fashion, then the build Status is shown as "queued" instead of "success". This defect will be fixed in coming fixpaks.

   However, when the Null Builder is invoked as part of a sequence of build actions in which there are other types of builders, then the status is properly handled. That is, the final status of the collector part is: success.

   Thus, even though it is incorrect, the current behavior is:

   - Now the View Information action shows that the buildStatus for archivo changed from:

     ```
     buildStatus      out_of_date
     ```

     To the new value:

     ```
     buildStatus      queued
     ```

   - Refresh the window and you will see that the icon for this part now has an icon that indicates that it was placed in a queue.

## PERFORMING BUILD EVENTS THAT AFFECT MANY PARTS

This section shows some build events that involve many parts at a time and that trigger other build events. This is the normal mode of using the Build function, in which the top build node (the collector part archivo.col) is selected and built, and the Build function will determine what parts, if any, need to be rebuilt. That is, you can avoid selecting every individual part and request the build action on them; only explicitly build the top build note and let TeamConnection take care of the rest.

## Touching archivo.h will mark all the other parts as out of date

This is a typical build scenario: once the build tree is up to date, then if one of the build dependencies is modified (that is, its time stamp is changed) then it is necessary to rebuild those parts that now are out of date.

For example, if the readme.txt file is changed, then only the zip part archivo.zip and the collector part archivo.col should be rebuilt, but the object code archivo.o and the executable code archivo should be left untouched because they are not affected by the change in readme.txt.

There are 2 actions that can cause a change in the time stamp of a part:

- Checking out and checking in a part in order to modify its contents.
- Touching a part to mark it as out of date.

  The contents of the part is left untouched.

  This is a similar concept of using the Unix utility "touch", in order to trigger build events.

  This is also functionally equivalent of checking out a part, not changing it, and then checking it in. The main difference is that the lastUpdate attribute is not changed with the touch operation.

  As you can guess, the touch action is faster than the checkout-checkin process.

Because the focus of this technical report is the Build function, only the technique of "touching" a part will be discussed, in which a change in a leaf dependency will trigger a chain reaction of build events.

In this case, the include header "archivo.h" will be touched. This causes the object code "archivo.o" to be marked as out of date, which in turn causes the executable "archivo" to also be marked as out of date. The chain reaction continues and the zip part "archivo.zip" and the collector part "archivo.col" are also marked as out of date.

1. Select a part, such as the include header "archivo.h" and perform Selected -> Touch ... and click OK.
2. Refresh the BuildView window and you will see that all the buildable parts have the black icon that indicates that they are out_of_date.
3. Select the collector part "archivo.col" and specify the Build action with the default options.
4. The Build Progress window will show the following messages that indicate that all the buildable parts in the build tree are being rebuilt.

```
6021-700 Number of distinct build events for this build: 4.
Build of 'archivo.o' started at '1998/11/04 05:32:41'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo.o' successfully completed at '1998/11/04 05:32:47'.
Completed Jobs: 1
Remaining Jobs: 3
Build of 'archivo' started at '1998/11/04 05:32:49'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo' successfully completed at '1998/11/04 05:32:54'.
Completed Jobs: 2
Remaining Jobs: 2
Build of 'archivo.zip' started at '1998/11/04 05:32:56'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo.zip' successfully completed at '1998/11/04 05:33:01'.
Completed Jobs: 3
Remaining Jobs: 1
Build of 'archivo.col' successfully completed at '1998/11/04 05:33:02'.
Completed Jobs: 4
Remaining Jobs: 0
Processing Completed for 'archivo.col'.
```

5. The View Build Messages window for every buildable part will be updated and will be practically the same as before, except for the time stamp.
6. Refresh the BuildView window and you will see that all the icons for the buildable parts have now the green check mark that indicate success.

## Forcing a build of archivo.col triggers a chain reaction of build events

In occasions, you may want to build the entire application regardless of the status of the dependency parts. In this case, you can select the higher build part and then build it with the Force option. This option will force the rebuild of every single part of the build tree:

1. Select the collector part "archivo.col" and specify the Build action.
2. From the Build Parts window, select the option "Forced". Then click OK.
3. The Build Progress window will show the following messages that indicate that all the buildable parts in the build tree are being rebuilt.

```
6021-700 Number of distinct build events for this build: 4.
Build of 'archivo.o' started at '1998/11/04 06:01:56'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo.o' successfully completed at '1998/11/04 06:02:01'.
Completed Jobs: 1
Remaining Jobs: 3
Build of 'archivo' started at '1998/11/04 06:02:03'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo' successfully completed at '1998/11/04 06:02:08'.
Completed Jobs: 2
Remaining Jobs: 2
Build of 'archivo.zip' started at '1998/11/04 06:02:10'
via a build agent on the host 'oem-ppc3.raleigh.ibm.com'.
Build of 'archivo.zip' successfully completed at '1998/11/04 06:02:16'.
Completed Jobs: 3
Remaining Jobs: 1
Build of 'archivo.col' successfully completed at '1998/11/04 06:02:17'.
Completed Jobs: 4
Remaining Jobs: 0
Processing Completed for 'archivo.col'.
```

4. The View Build Messages window for every buildable part will be updated and will be practically the same as before, except for the time stamp.
5. Refresh the BuildView window and you will see that all the icons for the buildable parts have now the green check mark that indicate success.

# MISCELLANEOUS TOPICS

## PASSING PARAMETERS WHEN DOING "TEAMC PART -BUILD"

The parameters for a part object being built will replace the parameters from the builder of the part object. This is useful in the following situations:

- For specific builds, such as for providing code that is enabled for debugging (such as using the '-g' flag for the C compiler in Unix).

  In Unix these flags are passed via environment variables that are created at the time the build is performed and they are made available to the process environment in which the build will be executed.

  These environment variables are specified in the field "parameters" from "teamc Part -build", for example:

  ```
  CFLAGS=-g
  ```

  In this case, the environment variable CFLAGS will be created with the specified value, such as '-g'. Then at build time, the build script (such as C-compiler.ksh) can read and use this environment variable, such as in the excerpts below:

  ```
  # Process environment variables
  flags=$CFLAGS

  # Display information (for tracking purposes)
  print "  Using flags:            $flags "

  xlc -c $source -o $target $flags
  ```

  In this case, the value for $flags in the C-compiler.ksh script will be '-g'.

  For more examples see “Examples of passing parameters as environment variables” on page 70.

- The builder parameters are correct for most parts but not correct for a few parts.

  You can use one builder for all of these parts and use part parameters to override the builder parameters when required. The part parameters can use variable substitution just like the builder parameters. You can also include the builder parameters in the part parameters by using $(BUILDERPARMS). The builder parameters will be substituted for $(BUILDERPARMS) in the part parameters.

  For more examples see “Examples of passing parameters using keyword substitution” on page 72.

## Examples of passing parameters as environment variables

In this section we will explore some aspects of the feature to pass parameters as environment variables when using the command "teamc Part -build" or the window Build Part:

- Specifying no parameters

  This is the case that was used previously in this document. The CFLAGS environment variable was not created and thus the script C-compiler.ksh did not use any flags, as shown in following lines from the build messages:

```
****************** Build Output Follows ************************
C-compiler.ksh: begin
  Processing source file: archivo.c
  Producing object file:  archivo.o
  Using flags:
C-compiler.ksh: end; return code=0
****************** End Of Build Output ************************
```

- Specifying parameters: CFLAGS=-g

  This is the correct way to specify a parameter (-g, which indicates to add debugging information to the object code during the compilation step) that is passed as an environment variable (CFLAGS) that builder has to obtain from the process environment.

  At build time, the script C-compiler.ksh will read the environment variable:

```
# Process environment variables
flags=$CFLAGS
```

  Then, the script will display and use the value:

```
# Display information (for tracking purposes)
print "  Using flags:            $flags "

xlc -c $source -o $target $flags
```

  The following lines from the build messages indicate the actual value that is used:

```
****************** Build Output Follows ************************
C-compiler.ksh: begin
  Processing source file: archivo.c
  Producing object file:  archivo.o
  Using flags: -g
C-compiler.ksh: end; return code=0
****************** End Of Build Output ************************
```

- Specifying parameters: -g

  If you omit the environment variable name and you specify only the value, then the build function will not create the environment variable and consequently the build script will not read it.

Thus, this case is identical to the case in which nothing specified at all in the parameters field.

- Specifying parameters: CFLAGS="-g -D__UNIX__"

  In this case the double quotes will ensure that several items are passed as a single value to the environment variable.

  The following lines from the build messages indicate the actual value that is used:

```
****************** Build Output Follows ************************
C-compiler.ksh: begin
  Processing source file: archivo.c
  Producing object file:  archivo.o
  Using flags: -g -D__UNIX__
C-compiler.ksh: end; return code=0
****************** End Of Build Output *************************
```

- Specifying parameters: CFLAGS=-g -D__UNIX__

  Because the double quotes were not used, only the value of '-g' is assigned to the environment variable and the rest of the string is ignored.

  The following lines from the build messages indicate the actual value that is used:

```
****************** Build Output Follows ************************
C-compiler.ksh: begin
  Processing source file: archivo.c
  Producing object file:  archivo.o
  Using flags: -g
C-compiler.ksh: end; return code=0
****************** End Of Build Output *************************
```

- Specifying parameters: CFLAGS=-g LFLAGS=-I/usr/include ZFLAGS=-r

  You can specify several environment variables with their values in the field parameters. These environment variables will be created and then it is up to the individual build scripts to handle them. In this case, the parameter field when building the collector part "archivo.col" may have the following:

```
CFLAGS=-g LFLAGS=-I/usr/include ZFLAGS=-r
```

  The C-compiler build script will show that the value for CFLAGS was used:

```
C-compiler.ksh: begin
  Processing source file: archivo.c
  Producing object file:  archivo.o
  Using flags:            -g
```

  The C-linker build script will show that the value for LFLAGS was used:

```
C-linker.ksh: begin
  Processing object files:   archivo.o
  Producing executable file: archivo
  Using flags:               -I/usr/include
```

The zipper build script will show that the value for ZFLAGS was used:

```
zipper.ksh: begin
  Processing input files: archivo readme.txt
  Producing target file:  archivo.zip
  Using flags:            -r
```

### Examples of passing parameters using keyword substitution

In this section we will explore some aspects of the feature to pass parameters by using keyword substitution:

## HINTS WHEN MODIFYING BUILDERS

During our testing of the Build function we modified the specification of the builders and here are some hints to keep in mind:

- When modifying the script for a builder, you need to specify the script name and the complete path for the file in the workstation. Do not assume that only the directory is sufficient, because the script name is NOT concatenated (appended) to the directory.
- The script name for a builder is the file name that will be used when the build script is extracted to the workstation. The recommendation is to use the same name for the script name and the builder name.
- When the builder is modified (or recreated, that is, deleted and then created), then the parts that use the builder are marked as out of date.
- In Unix, when modifying the parameters of a builder, you must enter the complete string again, and use the \$ sequence to enter environment variables such as '\$TC_INPUT'.

  If you just add one more to the end, the previous ones are ignored.

## WHAT HAPPENS WHEN A BUILD IS SUBMITTED

After a build is submitted to the family server the following step are carried out:

1. The family creates a build job: which parts need to be rebuilt, and in what order. Note that more than one part may need to be built, each by one build event.

   All events are in the pool specified in the part -build command, but they can be in different build environments depending on the environment attribute of the builder associated with the part.

2. The build event is queued until a build server is ready to do the build.
3. If a build server (teamcbld program) is found which is running in the pool and build environment for which a build event is ready, then the family server sends the necessary information to the build server for completing the build. The information sent includes:
   - The name and type of each input and dependent parts.
   - The name and type of the output parts.
   - The name and type of the build script.
   - The builder parameters, part parameters, and the parameters specified on the build.
   - The values of timeout, expected return codes.
   - The bulk contents, if any, of the input parts and the build script.

     Note that if the type of the builder is 'none', then the build script is expected to be on the build server's environment, for example a C compiler.

   The family server marks that build event as queued. At this point other build events in the job, which are not dependent on the first one, can be processed by other build servers.
4. The build server creates the environment in which to run the build script. This includes materializing the files on the local file system, setting environment variables and parameters.
5. Then the build script is invoked. The return code from the build script is used to determine success or failure of the build.
6. After the build script returns, the build results are sent to the family server. This includes the success/failure indicator and in the case of success, any output files.
7. After the family server receives the build results from the build server, it takes action based on the failure or success of the build event. If a build event fails, the remaining build events in the job are canceled unless -unconditional flag was specified for the part -build.

## MORE EXAMPLES OF BUILDERS FOR UNIX

These examples were provided by our co-worker Lee Perlov.

### zipbuild.ksh

```
#!/usr/bin/ksh
# $KW; $FN; $ChkD; $Ver; $EKW;
print VisualAge TeamConnection Version 3 zip file builder
print "TeamConnection environment variables:"
env | grep TC_
print "TeamConnection/DB2 environment variables:"
env | grep DB
```

```
print ""
print "Build Start"
print "@(#) Package context: Family $TC_FAMILY, Release $TC_RELEASE" > $$.tmp
print "@(#) Package context: Workarea $TC_WORKAREA" >> $$.tmp
zip -z $TC_OUTPUT $TC_INPUT < $$.tmp
if (( $? != 0 ))
then
  print -u2 "ERROR: zip command failed"
  rm -f $TC_OUTPUT
  exit 1
fi
rm $$.tmp
print '_____________________'
ls -l $TC_OUTPUT
print ""
print "Contents of zip file"
unzip -l $TC_OUTPUT
if (( $? != 0 ))
then
  print -u2 "ERROR: unzip command failed"
  rm -f $TC_OUTPUT
  exit 1
fi
print "Processing complete"
exit 0
```

**tarbuild.ksh**

```
#!/usr/bin/ksh
# $KW; $FN; $ChkD; $Ver; $EKW;
#
print VisualAge TeamConnection Version 3 tar file builder
print "TeamConnection environment variables:"
env | grep TC_
print "TeamConnection/DB2 environment variables:"
env | grep DB
print ""
print "Build Start"
tar -cvf $TC_OUTPUT $TC_INPUT
if (( $? != 0 ))
then
  print -u2 "ERROR: tar command failed"
  rm -f $TC_OUTPUT
  exit 1
fi
print '_____________________'
print 'Verification Process'
print ""
ls -l $TC_OUTPUT
print ""
```

```
print "Contents of tar file"
tar -tvf $TC_OUTPUT
if (( $? != 0 ))
then
  print -u2 "ERROR: tar command failed"
  rm -f $TC_OUTPUT
  exit 1
fi
print "Processing complete"
exit 0
```

# APPENDIX A. 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 |
|---|---|
| AIX, IBM, DB2, DB2 UDB, Universal Database VisualAge, TeamConnection, CMVC | IBM Corporation |
| Info-ZIP | Info-ZIP Group |
| Intel | Intel Corp. |
| Windows, Windows NT, | Microsoft Corporation |
| Unix | X/Open Co. Ltd. |

**END OF DOCUMENT**