mbuild Command

A proposal for a command to build and install software for Multics libraries.

Author: Gary Dixon
Date: October 25, 2019

Abstract

Multics presents programs and their supporting documentation to users as a set of program libraries. While Multics provides individual tools for preparing and documenting a program, no tool has sufficient data to easily install all program components into their appropriate library directories. This bulletin proposes a Build Script Language for describing components of a new program, or changes to an existing program; and an mbuild command that reads a Build Script file to compile, archive, bind, and prepare to install a program changeset into a target library.

Table 1: Revision History

Date	Revision	Author	Comment
2019-08-03	0.1	Gary Dixon	Initial draft.
2019-08-04	0.2	Gary Dixon	Correct example adding new_gate_ to use -rb 1 1 5. Correct example adding files to a private library to use -rb 4 5 5.
2019-08-06	0.3	Gary Dixon	Correct typo and grammar errors.
2019-08-07	0.4	Gary Dixon	Review comments from Eric Swenson on version 0.3.
2019-08-08	0.5	Gary Dixon	Responses to questions put to Eric Swenson on his version 0.3 comments.
2019-08-09	0.6	Gary Dixon	Enhanced for UNKNOWN library checks, and unchanged original-content segments.
2019-09-01	0.7	Gary Dixon	Fix minor typos.
2019-09-16	0.8	Gary Dixon	Add compare and hcom requests.
2019-09-17	0.9	Gary Dixon	Update tables forgotten in earlier version(s).
2019-09-21	1.0	Gary Dixon	Update install_ec examples for revised .mb.ec format.
2019-09-30	1.1	Gary Dixon	Fix minor bugs in structure comments, and text describing a few mbuild requests.
2019-10-25	1.2	Gary Dixon	Fix inconsistency in Build Script Language BNF definition.

Table of Contents

List of Tables

Introduction

Multics presents programs and their supporting documentation to users as a set of program libraries. Each library consists of: a directory of programs (executable object segments); a directory of documentation (info segments); and source, object, and include directories containing files that produce the executable programs. Policies and procedures govern contents and naming within each library directory.

While Multics provides individual tools for preparing program components and documenting a program, no tool exists to facilitate the assembly, preparation, and installation of all these components. In fact, no tool ever gathers an overview of all program components sufficient to supervise their preparation and installation.

Past installers have used personal exec_coms to automate frequent installation scenarios. However, none of these was so effective to warrant installing the exec_com as a standard library tool. Their lack of success stems from complexities of Multics library environment: the many segment types to build/install; a wide variety of software development tools; several different libraries to maintain, each with its own build/install requirements; lack of a Multics-wide product packaging strategy; few tools to implement and enforce installation policies and procedures.

This bulletin addresses some of these deficiencies. It defines a Build Script Language for describing components of a new program, or changes to an existing program. It proposes a new subsystem called mbuild that: scans an installation directory looking for changeset components; constructs a Build Script file describing the changeset; asks the library maintainer to review, and perhaps edit, this script; then follows the approved script to compile, archive and bind, and prepare a Build_exec_com that directs update_seg to install components of the changeset into the directories of a target library.

Describing a Program Changeset

In its simplest form, a program consists of:

- **source**: program algorithms expressed in some source programming language;
- **includes**: additional source excerpts included by the program source; and
- **info**: optional info segment(s) documenting use of the program.
- **object**: executable segment produced by compiling (translating) the source program and include files.

Library policy recommends grouping executable programs that perform similar functions or work together as a subsystem. Groups are formed by binding the similar executables into a single bound object segment. Such grouping introduces the following:

- **unbound object**: name used for the executable segment produced by compiling a single source program and its include files, as described above. This qualified name distinguishes this standalone object from…
- **bound object:** related unbound object programs grouped into a single executable segment; also called the *bound object segment* or *bound segment*.
- **bind file:** instructions for grouping several unbound objects into a single bound object.
- **source archive:** source files that produce the programs forming a bound object, grouped as components of an archive segment.
    - Include files are not placed in these source archives. They remain as separate segments in the include directory, where they may be easily included by other source programs.
- **object archive:** the unbound object programs forming a bound object, plus any bind file, grouped as components of an archive segment.

Library conventions for naming segments which produce a bound object:

A. A bound object has a primary name in the form: bound_XXX_, where: bound_ is a constant prefix; XXX is a variable string that identifies content/purpose of the bound segment; and the final _ distinguishes the bound_XXX_ name from a command name (since command names do not end with an underscore).
B. A source archive for bound_XXX_ has a primary name: bound_XXX_.s.archive.
C. An object archive for bound_XXX_ has a primary name: bound_XXX_.archive.
D. If the sources forming a bound segment are too large to fit within the max length of a single source archive segment, they may be split among several source archives whose name contains a 1-digit sequence number: bound_XXX_.1.s.archive, bound_XXX_.2.s.archive, …
E. The unbound objects generated by compiling source of each source archive are stored in a correspondingly named object archive: bound_XXX_.1.archive, bound_XXX_.2.archive, …
F. The bind file bound_XXX_.bind for a program that has multiple object archives is always stored in bound_XXX_.1.archive.

Build Script Language

The terms defined above form the basis for a language that describes operations grouping components into a new program; or operations changing components of an existing program.

Statements in this Build Script Language begin with: a label identifying the type of component; its segment name; the library or library.directory in which it is installed; and the operation being performed on that segment. In the example shown below, constant labels are shown in **BOLD** text to distinguish them from file name, library name, and other strings that vary. However, an actual build script file contains no bold text.

```
Unbound_obj: volume_dump_switch_on      IN: sss  REPLACE;
   source:   volume_dump_switch_on.pl1         REPLACE;

Bound_obj:   bound_library_tools_       IN: tools      UPDATE;
  bindfile:  bound_library_tools_.bind             REPLACE;
   source:   library_pathname.pl1                  REPLACE;

Info:        library_pathname.info      IN: privileged.info    REPLACE;
               add_name: lpn.info;

Include:     lib_descriptor_.incl.pl1   IN: include    REPLACE;
```

Each capitalized label (**Unbound_obj**, **Bound_obj**, **Info**, **Include**) identifies the type of a major item being installed in a library. Lowercase labels following a major item identify component files (**source**, **bindfile**, **source_arch**) composing a bound segment; or names to be added to or deleted from the major item. A **source_arch** statement is needed only for a **Bound_obj** having more than one source archive, to indicate which source archive contains subsequent **source** components.

The operation being performed for each item of the changeset is given near the end of its statement.

- An **Unbound_obj**, **Include**, or **Info** segment may have: **ADD REPLACE DELETE**.
- A **Bound_obj** may have: **ADD UPDATE DELETE**, where **UPDATE** indicates that:
  - One or more component items is changing with operation: **ADD REPLACE DELETE**.
  - **source_arch** is changing with operation: **ADD UPDATE**.

An **add_name** or **delete_name** statement may be given below an **Unbound_obj** or **Info** statement. Names on a **Bound_obj** are controlled by statements in its bind file. **Include** statements may only have one name.

A build script file may optionally begin with **Description:** followed by one or more lines of text, ending with an **Installation_directory:** line.

An example of an actual build script file is shown below. Information about compiler to be used for a given source may be present, along with compile options; the user may edit the default compile options suggested in a computer-generated build script file.

```
Description:
Test update of standard bound object, including:
 - bindfile
 - source component
 - info segment

Installation_directory:  >user_dir_dir>Multics>GDixon>w>MCR002;

 Build_script:         MCR002.mb;

 Bound_obj:            bound_library_tools_          IN: tools  UPDATE;
     bindfile:         bound_library_tools_.bind      REPLACE;
     source:           library_pathname.pl1           REPLACE    compiler: pl1 -ot;

 Info:                 library_pathname.info         IN: privileged.info  REPLACE;
                         add_name:  lpn.info;
```

The **Description**, **Installation_directory** and **Build_script** statements are optional, but are usually supplied in a computer-generated build script file.

- **Description**: is a multi-line text field in which the installer describes the purpose of the change, MCR authorization information, possible target release, etc.
  - The installer can add a Description field to the Build_script file using an editor; or can use the mbuild set -description request which prompts for the multi-line text.
  - mbuild passes this description to the update_seg -log argument. The multi-line text ends on the line preceding the Installation_directory: label.
  - update_seg includes this string to describe the installation in its Installations.info and Installation.log files. (See the mbuild -set_log_dir control argument; or the set -log_dir request.)

- **Installation_directory**: gives the absolute pathname of the directory in which mbuild was invoked, a directory containing the original content files to be installed. Those files are described by remaining Build_script statements.
  - This pathname is useful if content of the Build_script is mailed to others for comment or review; or if Build_script files are permanently saved to document the changeset.

- **Build_script**: names the file containing the build script statements as an original-content file residing in the installation directory.
  - When the mbuild save request saves this script, this file becomes part of the changeset, and is therefore listed in the build script. Though it is not installed, mbuild recognizes its purpose as data describing the changeset.

The full Build Script Language is documented in BNF-form in Appendix A of this MTB. More examples of build scripts are given in the sections describing the mbuild subsystem.

Requirements for a Build/Install Tool

A user preparing a new set of programs must be familiar with the wide variety of tools for developing software on Multics.

Table 2 lists the most used tools.

REQUIREMENT 1: A generic build tool should know when to use these tools to implement the build/install process, and how to use each tool.

Table 2: Multics Software Development Tools

COMPILERS	GROUPING	LIBRARY MAINTENANCE
pl1	archive, ac	update_seg, us
alm	archive_sort, as	library_info, li
create_data_segment, cds	archive_table, act	library_descriptor, lds
reductions, rdc	bind, bd	library_fetch, lf
lisp_compiler, lcp		library_pathname, lpn
library_descriptor_compiler, ldc		peruse_crossref, pcref
pl1_macro, pmac		
PROGRAMMING / AUDIT	**DOCUMENTATION**	
compare_ascii, cpa	help	
history_comment, hcom	validate_info_seg, vis	
	compose, comp	
	add_pnotice	

The compilers translate source programs and include files into an unbound object. Some compilers employ a two-step process: step 1 translates a program written in a highly-specialized language to code written in a general-purpose language; step 2 invokes the general-purpose compiler to translate this intermediate code into an executable unbound object.

Table 3 shows some of these two-step compilers. In most cases, the step 1 translator automatically invokes the step 2 compiler when step 1 completes. In some cases, the intermediate language source file remains after step 2 completes; or remains if step 2 encountered errors.

REQUIREMENT 2: A generic build tool should know how to invoke two-step translators and manage any intermediate source files left behind.

Table 3: Two-step Compilers

Kind of Translator	File Suffix	Specialized Translator	Intermediate Language	General-purpose Compiler	Unbound Object
data segment source	.cds	cds	PL/I	pl1	data table
reduction-based translator	.rd	rdc	PL/I	pl1	translator
macro-based source	.pmac	pmac	PL/I	pl1	object segment
library descriptor table	.ld	ldc	ALM	alm	data table

Multics uses several libraries to hold different kinds of executable objects. These are summarized in Table 4. Each library is divided into directories that hold different types of segments. Some directories are shared between libraries. Table 5 describes these directories. Table 6 shows which directories are defined for each library, and how they are shared between libraries.

REQUIREMENT 3: A generic build tool should understand this library structure, and target certain segment types for install into a particular directory of a selected library.

Table 4: Multics Libraries

Library Name	Purpose
sss	Holds programs needed by most users.
tools	Holds programs needed by software developers and system maintainers.
hard	Holds programs that are part of the Multics supervisor, and shared by all users.
unb	Holds software products sold individually to Multics sites.
mcs	Holds communications software that runs on Multics Front-end Network Processors (FNPs).

Table 5: Directory Types in Multics Libraries

Directory Name	Purpose
source	Holds source files for unbound objects; and source archives for bound objects.
include	Holds source include files referenced by (incorporated into) source files.
object	Holds object files for unbound objects; and object archives for bound objects.
execution	Holds executable files (bound or unbound objects) and data for programs available to users.
info	Holds info segments describing the executable files and program data.
i	Holds control files and maps showing how hardcore and communications software are linked into a bootable operating system.
listings	Holds compilation listings for individual sources; and bind listings describing how unbound objects were bound together into a bound object. Most sites do not generate or store listings, since they require large amounts of disk storage.

Table 6: Directories in Each Multics Library

Library	**Directory**	**Pathname**	**Comment**
sss	source	>ldd>sss>source	
	include	>ldd>include	Shared by all libraries.
	object	>ldd>sss>object	
	execution	>sss	
	info	>doc>info	Shared by sss, tools, unb.
tools	source	>ldd>tools>source	
	object	>ldd>tools>object	
	execution	>tools	
privileged	info	>doc>privileged	Shared by tools, unb
unb	source	>ldd>unb>source	
	object	>ldd>unb>object	
	execution	>unb	
hard	source	>ldd>hard>source	
	object	>ldd>hard>object	
	execution	>ldd>hard>execution	
	i	>ldd>hard>info	
mcs	source	>ldd>mcs>source	
	object	>ldd>mcs>object	
	i	>ldd>mcs>info	

In addition to executable programs, and their source and object components, the Multics Libraries include a variety of data tables, command files (exec_coms), data archives, etc. Each type of segment is installed in a particular library directory, following an install model designed for that type of segment. Table 7 lists build/install paradigms currently used in the Multics Libraries.

REQUIREMENT 4: A generic build tool should identify each segment type by its name suffix, and map that type to a particular build/install model or paradigm. The tool should report an error when asked to install an unknown segment type, or when given a known segment type that follows an unsupported build/install paradigm.

Table 7: Build/Install Paradigms

Paradigm Name	Purpose	Example
source	Compile source; add/delete/replace source and object components in archives.	v2pl1.pl1
bindfile	Add/replace/delete .bind file in object archive.	bound_pl1_.bind
source_arch	Install source archive into source directory.	bound_pl1_.**.s.archive
object_arch	Bind and install object archives into object directory.	bound_pl1_.**.archive (except those ending in .s.archive)
Bound_obj	Install executable bound segment into execution directory.	bound_pl1_
Unbound_obj	Install executable unbound segment into execution directory.	bisync_, installation_gate_, edm
target_only	Install segment into info or include directory.	info segs, include files.
listing	Install .list file into listings directory.	v2pl1.list, hcs_.list
source_x_only	Install source or unbound source archive into source and execution directories.	admin.ec, tss_basic_.archive
object_x_only	Install segment into object and execution directories.	TTF.ttf, pl1.dcl

Segments already installed in the Multics Libraries provide information about library in which they are installed, bound segment of which they are a component, etc.

REQUIREMENT 5: A generic build tool should try to locate each segment being installed; if found, the matching segment's library, directory, and containing archive (if any) provides data for a REPLACE operation. For REPLACE operations, the tool should fabricate information needed in the Build Script file. For DELETE operations, the user should only need to give the segment's type, name, and operation.

Build and Install Toolset

This bulletin proposes:

- a new mbuild subsystem to perform build and install tasks;
- supported by a new mbuild_info_.cds database to identify segment types and their build paradigms, and an mbuild_type command to display that database; and
- a small change to the existing library_pathname command/AF allowing it to always return pathnames of existing segments in the Multics Libraries, beginning with a fixed library.dir root pathname.

The last of these proposed changes is simplest to describe.

Change to library_pathname

The library_pathname command/AF is designed to search one or more library directories for segments matching a starname, returning pathnames of matching library segments or archive components. That mechanism could be used to meet REQUIREMENT 5 above. But it does not work quite as needed.

library_pathname active function always returns the shortest possible pathnames. While short pathnames are useful in some situations, an mbuild command trying to locate an existing program source within the Multics Libraries needs pathnames that begin with one of the known library.directory root pathnames defined by the multics_libraries_.ld descriptor. This permits the pathname to be mapped directly onto a unique library.directory identifier given in the descriptor.

To retain the current operation of library_pathname for most users, the normal library_pathname$library_pathname entrypoint will continue returning shortest pathnames. A new library_pathname$mbuild entrypoint will be added that avoids calling get_shortest_path_.

A new mbuild_lpn command will permit the library maintainer to easily see exact pathnames returned by library_pathname$mbuild. Inside the new mbuild subsystem, the lpn request/active request will call mbuild_lpn$lpn_request to provide this same information.

New Build Information Database

REQUIREMENT 4 above calls for a method to map a segment name onto a segment type, which specifies how to build/install that segment in the Multics Libraries (the segment's build paradigm). Add a new mbuild_info_ data segment to implement this requirement. This data segment will consist of 4 arrays of structures:

- An ordered list of seg_type_info structures, each containing a starname to compare with the name of any segment being installed. The first matching starname ties the segment being installed to that structure's type_ID and build_paradigm values.
- An array of build_paradigm_info structures, each naming one of the build paradigms given in Table 7 above, describing its purpose, giving a few example segments that follow the paradigm, and ending with suggested steps to follow to build/install segments adhering to that paradigm.

- The third and fourth arrays in mbuild_info_ identify threaded lists of segments that will be constructed by the mbuild system, as it builds a changeset. These are useful mainly to display mbuild threaded lists when debugging the subsystem.

The mbuild_info_.incl.pl1 include file declares details of these structure arrays. Each level 2 element begins one of the arrays of structures outlined above.

```
dcl mbuild_info_$seg_types fixed bin external static;

dcl 1 mbuild_info                   aligned based (addr (mbuild_info_$seg_types)),
      2 seg_types,
        3 segN                      fixed bin,
        3 seg_type_info             (0 refer (mbuild_info.seg_types.segN)),
          4 type_ID                 fixed bin,          /* numeric identifier for this seg_type array entry.      */
          4 source_starname         char (32) var,      /* e.g., **.pl1, **.incl.pl1, bound_*.**.s.archive        */
          4 description             char (60) var,      /* content of segs having this source_suffix.             */
          4 mbuild_type             char (20) var,      /* mbuild Seg(<type>) field value for this segment type.  */
          4 build_paradigm          fixed bin,          /* build/install paradigm for this source_suffix          */
          4 compiler                char (32) var,      /* name of compiler/translator for this seg (if any)      */
          4 default_compile_options char (32) var,      /* options used when building to install in Multics libs. */
          4 intermediate_suffix     char (12) var,      /* suffix of intermediate file generated by compiler      */
          4 object_suffix           char (12) var,      /* suffix of seg generated by compiler                    */
          4 default_library         char (32) var,      /* default library which holds this type of file          */
                                                        /*  (used when adding new files).                         */
      2 bld_paradigms,
        3 paradigmN                 fixed bin,
        3 bld_paradigm_info         (0 refer (mbuild_info.bld_paradigms.paradigmN)),
          4 name                    char(32) var,       /* ex: compile, source_x_only, ...                        */
                                                        /*  Names never contain periods or spaces.                */
          4 purpose                 char(76) var,       /* what does this build_type do.                          */
          4 examples                char(70) var,       /* typical seg types using this paradigm.                 */
          4 steps                   char(1600) var,     /* steps involved in paradigm                             */

      2 thread_selectors,
        3 selectorN,
        3 selector_info             (0 refer (mbuild_info.thread_selectors.selectorN)),
          4 sel_ID                  char(16) var,       /* selector ID string                                     */
          4 sel_value               fixed bin,          /* selector value: one of the following constants:        */
      2 thread_summary,
        3 summaryN fixed bin,
        3 selector_summary          (0 refer (mbuild_info.thread_summary.summaryN)) char(32) var;
                                                        /* Summary of sel_ID possible values for error msg.       */

dcl (PDM_source                     init(1),            /* Compile, schedule bind, if part of bound seg;          */
                                                        /*  otherwise, install in ldd and x dirs.                 */
     PDM_bindfile                   init(2),            /* Bind instructions, only found in bound obj archive.    */
                                                        /*  (ex: bound_pl1_.bind)                                 */
     PDM_source_arch                init(3),            /* No compile, just install in ldd source dir.            */
                                                        /*  (ex: bound_pl1_.**.s.archive)                         */
     PDM_object_arch                init(4),            /* Bind a bound seg; install source/object archives in    */
                                                        /*  ldd; install bound object in x dir.                   */
     PDM_Bound_obj                  init(5),            /* Install executable bound segment in x dir.             */
                                                        /*  (ex: bound_pl1_)                                      */
     PDM_Unbound_obj                init(6),            /* Install unbound object in x dir.                       */
                                                        /*  (ex: hcs_ compiled from hcs_.alm)                     */
     PDM_target_only                init(7),            /* No compiles; just install in target dir.               */
                                                        /*  (ex: info segs, include files, bound_xxx_.s.archive)  */
     PDM_listing                    init(8),            /* No compiles; just install in listing.                  */
                                                        /*  (ex: pl1.list, probe.list, hcs_.list)                 */
     PDM_source_x_only              init(9),            /* No compile, just install in ldd source and x dirs.     */
                                                        /*  (ex: admin.ec, >ldd>unb>s>tss_basic_::*.basic)        */
     PDM_object_x_only              init(10),           /* No compile or archive; install in ldd object and x.    */
                                                        /*  (ex: TTF.ttf, pl1.dcl)                                */
     PDM_mbuild_support             init(11),           /* No compile or archive or install; support file created */
                                                        /*  or used by the mbuild subsystem.                      */
                                                        /*  (ex: MCR10056.mb, MCR10056.mb.il)                     */
```

Data for these arrays is defined in an mbuild_info_.cds data segment. The data object is a component of bound_mbuild_.

mbuild_type Command/AF

A new command/AF displays information from the first two arrays.

```
help mbuild_type -bf

Syntax as a command:  mbt OPERATION {name} {-control_args}

Syntax as an active function:
  [mbt OPERATION name {-control_args}]

List of operations:
paradigm, pdm   seg_type, seg
```

The seg_type operation of this command maps a given segment name onto segment type information; or displays starname matching segments that follow a given build paradigm.

```
help mbuild_type.seg_type -a
>user_dir_dir>Multics>GDixon>info>mbuild_type.seg_type.info   (56 lines in info)
2019-06-10  mbt seg_type operation.

Syntax as a command:  mbt seg {seg_name} {-control_args}
                      mbt seg -for_paradigm paradigm_name

Syntax as an active function:
  [mbt seg seg_name {-control_args}]

Function:  displays information about the different types of segments
installed in the Multics Libraries.

If invoked without args, summarizes known segment types installed in
the Multics Libraries.  This includes a starname selecting a group of
segments, and characteristics of segments in the group.

If invoked with a seg_name and no control_args, displays full
information about building and/or installing that segment.

As an active function, returns selected information about build a given
segment.

Arguments:
operation
   is "seg_type" or "seg"
seg_name
   names a segment whose build/install information is displayed.  This
   argument is optional when invoked as a command, but required when
   used as an active function.

Control arguments:
-for_paradigm paradigm_name, -fpdm paradigm_name
   selects all segment types built using the given paradigm.  This
   control argument may not be used as an active function.
-all, -a
   displays all information for the build type.
-name, -nm
   displays a starname that matches segments in the build type.
-description, -desc
   displays a title for segments matching the starname.
```

```
-type, -tp
   displays the <seg-type> value given by mbuild in its
   Seg(<seg-type) structures for segments having the build type..
-library, -lb
   displays library in which this type of file is installed.

-compiler
   displays compiler command name used in building the segment.
-option, -op
   displays default compiler option(s) used in building the segment.
-isuffix, -isfx
   displays suffix of any intermediate files generated by the compiler.
   For example, the reductions compiler: generates a .pl1 file; invokes
   pl1 to compile this file; and leaves the .pl1 file in working
   directory as a debugging aid.
-osuffix, -osfx
   displays suffix (if any) of object segment generated by the
   compiler.

-paradigm, -pdm
   displays name of the paradigm used in building the segment.
-steps
   displays steps performed as part of this build paradigm.
```

The paradigm operation displays information about one or more build/install paradigms.

```
help mbuild_type.paradigm -a
>user_dir_dir>Multics>GDixon>info>mbuild_type.paradigm.info   (35 lines in info)
2019-06-10  mbt paradigm operation.

Syntax as a command:  mbt pdm {paradigm name} {-control_args}

Syntax as an active function:
  [mbt pdm paradigm_name {-control_args}]

Function: displays information about build paradigms, the procedures
and policies for building a given type of segment, and installing it
into the Multics Libraries.

If invoked without args, displays a 3-line description of the known
build paradigms including: name, purpose and example segments using
that paradigm.

If invoked with a paradigm_name and no control_args, displays a longer
description of that paradigm.

As an active function, specific information about a given paradigm is
returned.

Arguments:
operation
   is "paradigm" or "pdm".
paradigm_name
   names a paradigm to be displayed.  This is optional when invoked as
   a command, but required when used as an active function.

Control arguments:
-all, -a
   displays all information for the paradigm.
```

```
-name, -nm
   displays the paradigm name.
-purpose
   displays the purpose of the paradigm.
-example
   displays an example segment that is built or installed using the
   paradigm.
-steps
   displays steps performed as part of this build paradigm.
```

The two operation modes of the mbuild_type command are available within the new mbuild subsystem as requests: seg_type, seg and paradigm, pdm

An example of mbuild_type seg output shows information provided for building and installing the segment: v2pl1.pl1.

```
mbuild_type seg v2pl1.pl1

----------  v2pl1.pl1
source_starname:              **.pl1
description:                  PL/I Source File
mbuild_type:                  source
default_library:              source
build_paradigm:               source
 I. For replace/add of source:
    1. Compile source.
    2. If source is part of bound_xxx_.**.s.archive:
       a. Update source in its .s.archive.
       b. Update compiled object in corresponding .archive.
       c. update_seg add/replace changed source archive in >ldd>LIBRARY>s.
       d. Schedule bind of object archives comprising bound_xxx_.
    3. Else for non-archived source:
       a. update_seg add/replace source in its >ldd>LIBRARY>s dir.
       b. update_seg add/replace derived object in >ldd>LIBRARY>o and >LIBRARY.
 II. For delete of source:
    1. If source is part of bound_xxx_.**.s.archive:
       a. Delete source from bound_xxx_.**.s.archive.
       b. Delete derived object from corresponding object archive.
       c. update_seg replace changed source archive in >ldd>LIBRARY>s.
       d. update_seg replace changed object archive in >ldd>LIBRARY>o.
       d. Schedule bind of changed object archive.
    2. Else for non-archived source:
       a. update_seg delete the source in >ldd>LIBRARY>s.
       b. update_seg delete derived object in >ldd>LIBRARY>o and >LIBRARY.
compiler:                     pl1
default_compile_options:      -ot
object_suffix:
```

The initial array of segment types is shown below. As the mbuild subsystem is used to install a wider variety of segments, more segment type array entries can be added to this list.

```
mbuild_type seg -nm

Build can identify segments matching these starnames:
  bound_*
  *
  bound_*.bind
  bound_*.**.s.archive
  bound_*.**.archive
  **.incl.*
  **.pl1
  **.alm
  **.lisp
  **.cds
  **.ld
  **.rd
  **.pl1.pmac
  **.mb.ec
  **.mb.il
  **.mb.io
  **.ec
  **.info
  **.mb
  **.list
  **.dcl
  **.ttf
  **.archive
```

mbuild_type can also display segments sharing the same build paradigm, and selectively display information about those segment types.

```
mbuild_type seg -for_paradigm source -name -desc -compiler

source_starname:              **.pl1
description:                  PL/I Source File
compiler:                     pl1
----------
source_starname:              **.alm
description:                  ALM (assembler) Source File
compiler:                     alm
----------
source_starname:              **.lisp
description:                  LISP Source File
compiler:                     lisp_compile
----------
source_starname:              **.cds
description:                  Data Segment Source File
compiler:                     cds
----------
source_starname:              **.ld
description:                  Library Descriptor Source File
compiler:                     ldc
----------
source_starname:              **.rd
description:                  Reduction Language Source File
compiler:                     rdc
----------
source_starname:              **.pl1.pmac
description:                  PL/I Source File with Macro Definitions
compiler:                     pmac
```

Build and Install Subsystem: mbuild

The process for building and installing segments into the Multics Libraries may be divided into a sequence of steps. Starting with an installation directory containing original source segments and their supporting files, this sequence might include steps shown in Table 8.

Table 8: Steps in the Build/Install Process

a. Scanning the directory to get a list of segments.
b. Looking in Multics Libraries to see if earlier versions are already installed in a library, thereby determining whether each segment is being added or replaced.
c. Determining whether each segment is a member of a bound segment.
d. Organizing the segments by location within the libraries, and by name.
e. Determining whether each segment is a source segment that must be translated or compiled into an object; that is, determining each segment's build paradigm.
f. Organizing segments by their build paradigm, to schedule the build steps appropriate to each paradigm.
g. Compiling segments that require translation, and tracking the success and outputs of each compilation.
h. Grouping members of a bound segment into source archives, and object archives.
i. Binding components of a bound object archive into the bound object segment.
j. Issuing commands to install both original-content segments and their derived-content objects into the Multics Libraries.
k. Cleanup after the installation (to remove derived-content objects, and reduce storage space used by the installation directory). This could involve deleting: the entire directory; or remnants of the bind or install parts; or all derived-content segments, etc.

The actual steps required for a given build/install task depend upon the build paradigm of each segment to be installed. Steps may not be needed for some segment types. Some steps may have to be repeated if errors are encountered while building or installing.

Also, the steps must allow for input from the software developer and/or the installer to:

- record maintenance information supplementing the installation (Multics Change Request number authorizing the installation; overall purpose of the installation, etc.);
- specify where new segments are to be stored in the libraries;
- identify segments requiring special compiler options;
- add or remove names from derived-content segments;
- delete existing segments from the library.

Mindful that past attempts to standardize the build and install mechanism were only partially successful, this latest attempt at a build tool will include a few innovations.

- Use an implementation platform more flexible, more powerful, and easier to use than exec_coms. The latest subsystem utilities (ssu_) infrastructure best matches this platform requirement.
- Include a mechanism for tracking a segment to be installed in many different ways: as a segment known to mbuild; as a segment residing in the installation directory; as a source segment to be compiled; as a segment output by a compilation; as a component of a source or object archive; as a bound or unbound object to be installed; etc. A new threaded-list mechanism will be created to meet this requirement.

mbuild Implemented as an ssu_ Subsystem

An ssu_ subsystem is composed of requests that the user may invoke as needed. These request programs are linked by a common data structure maintained by ssu_ and passed to each request program. This infrastructure is ideal for implementing one or more steps from Table 8 as an ssu_ request. By issuing a sequence of requests, the installer can guide and monitor the build/install tasks; and change inputs or repeat steps as needed. Subdividing the work into a sequence of requests allows mbuild to capture and report on build progress as the work proceeds; and to recommend the next request appropriate to the segments being installed.

Various ssu_ features support the installation effort:

a. code for basic requests needed in many subsystems: list_requests, ?, help, quit.
b. support for per-request info segments describing operation of each request.
c. adding Multics commands (e.g., compilers, bind, library_fetch, etc.) as requests invocable inside the subsystem.
d. interruptability of long-running build requests (e.g., compiling all the source segments) if a serious error occurs, followed by re-entry into the subsystem via the program_interrupt (pi) command.
e. support for per-subsystem abbreviations.

The main requests proposed for mbuild are described in subsequent sections of this bulletin.

NOTE: item (c) in the list above points out that mbuild invokes many Multics commands, compilers, library maintenance tools, etc. These external commands have been added to the mbuild_request_tables_.alm list of requests, and are invoked as “hidden requests”. Such supporting utilities are hidden so they do not clutter up the list_requests (lr) output with commands the installer will seldom invoke directly. However, “list_requests -all” does show these commands; and the mbuild help request does display their standard command info segments. If future changes to mbuild support new compilers, or additional utilities (e.g., history_comment, compare_ascii, etc.), such tools may have to be added to the mbuild_request_tables_.alm as hidden requests.

The following Multics command/AFs are included as mbuild hidden requests/active requests:

```
archive, ac               add_name, an         home_dir, hd
archive_table, act        delete_name, dn      print_wdir, pwd, working_dir, wd
bind, bd                  file_output, fo      process_dir, pd
compare_ascii, cpa        revert_output, ro
history_comment, hcom
library_fetch, lf

alm
create_data_segment, cds
library_descriptor_compiler, ldc
lisp_compiler, lcp
pl1
pl1_macro, pmac
reductions, rdc
```

mbuild Threaded-List Mechanism

Multics code includes many ways of linking structures into a single threaded list. None of these methods is easily expanded to permit the same structure to be threaded into multiple lists at the same time. mbuild needs such functionality to change its view of segment relationships depending upon the build step underway at the moment. It must also maintain a history of such views, so that earlier tasks may be repeated if necessary; threading a segment's structure into one list at a time loses this history of earlier lists including the segment.

The new mechanism is called mbuild_Tlist_ (threaded-list). It provides a one-to-many cardinality that links a list anchor point (list head/tail pointers) to an ordered sequence of structures.

The list anchor, called a Tlist_base is declared in mbuild_Tlist_dcls_.incl.pl1:

```
 dcl 1 Tlist_base aligned based, /* Structure anchoring the threaded list of items. */
     2 headP ptr,                /* pointer to first item in the list.              */
     2 tailP ptr,                /* pointer to last item in the list.               */
     2 holderP ptr;              /* pointer to structure holding this Tlist_base.   */
```

The anchor is referenced as an element of a structure, such as the build_data structure passed among all the mbuild requests. Its headP and tailP point to a Tlist_data structure residing in each structure threaded onto the list.

```
 dcl 1 Tlist_data aligned based, /* Structure contained in items threaded together. */
      2 prevP ptr,               /* pointer to prev element of same type in the list*/
      2 nextP ptr,               /* pointer to next element of same type in the list*/
      2 itemP ptr,               /* pointer to containing structure item.           */
      2 baseP ptr;               /* pointer to list base (head/tail pointers).      */
```

Operators on these lists are defined in mbuild_Tlist_.incl.pl1. Each is a PL/I quick procedure: a code block that looks like a subroutine, but is invoked without pushing a stack frame onto the stack. Each structure on the list has both prevP and nextP pointers, allowing walking through the list in forward or backward directions.

A threaded-list example is the build_data.Seg_Tb list, which threads together all segment (Seg) structures created by mbuild.

```
dcl 1 build_data aligned,
    ...,
    2 Seg_Tb aligned like Tlist_base,
    ...,
```

This list is initialized by the call to the Tlist_initialize quick procedure, defined in mbuild_Tlist_.incl.pl1:

```
     call Tlist_initialize (addr(build_data.Seg_Tb), addr(build_data));
```

This call sets build_data.Seg_Tb pointers = null(); and Seg_Tb.holder = addr(build_data).

A structure that might be threaded onto this list declares a correspondingly-named Tlist_data element called Seg.Seg_Td:

```
dcl 1 Seg aligned based(SegP),
    ...,
    2 Seg_Td aligned like Tlist_data,
    ...,
```

When an instance of this structure is allocated, its Seg_Td member has all its pointers initialized to null():

```
Seg.Seg_Td = Tlist_initial_data;
```

The Seg instance is added to the tail of the build_data.Seg_Tb threaded list by a call:

```
call Tlist_add_to_tail (addr(build_data.Seg_Tb), SegP, addr(Seg.Seg_Td));
```

This sets Seg_Td.itemP = SegP; Seg_Td.baseP = addr(build_data.Seg_Tb); and Seg_Td.prevP to the last item on the list before this call was made. Seg_Td.nextP = null().

Typical code for walking forward through list items is coded as a PL/I do-repeat block:

```
do SegP = Tlist_first_item (addr(bld.Seg_Tb))
   repeat Tlist_next_item  (addr(Seg.Seg_Td))  while (SegP ^= null());
     ...
     end;
```

The mbuild_Tlist_.incl.pl1 include file defines operations on threaded lists as shown in Table 9 below.

Table 9: mbuild_Tlist_ Threaded-List Operations

Subroutine	Function Performed
Tlist_initialize (list_baseP, holderP);	Initialize Tlist_base element of holderP structure.
Tlist_holder (list_dataP);	Return pointer to structure holding threaded list anchor.
Tlist_member (list_dataP);	Returns True if list_data is an active member of a threaded list.
Tlist_is_empty (list_baseP);	Returns True if there are no members on list_baseP anchor.
Tlist_add_to_head (list_baseP, itemP, item_list_dataP);	Adds itemP to head of list anchored by list_baseP.
Tlist_add_to_tail (list_baseP, itemP, item_list_dataP);	Adds itemP to tail of list anchored by list_baseP.
Tlist_add_insert_before (list_baseP, new_itemP, new_item_list_dataP, cur_item_list_dataP);	Adds new_itemP before cur_item_list_dataP.itemP in the threaded list.
Tlist_add_insert_after (list_baseP, new_itemP, new_item_list_dataP, cur_item_list_dataP);	Adds new_itemP after cur_item_list_dataP.itemP in the threaded list.
Tlist_first_item (list_baseP);	Returns pointer to first item in list.
Tlist_last_item (list_baseP);	Returns pointer to last item in list.
Tlist_prev_item (item_list_dataP);	Returns pointer to previous item in list.
Tlist_next_item (item_list_dataP);	Returns pointer to next item in list.
Tlist_remove (item_list_dataP);	Removes item from list.
Tlist_count (list_baseP);	Returns count of items in the list.

The mbuild Command

Work on a build/install effort begins with an installation directory, in which either the developer or installer has placed all original-content segments to be installed.

The installer changes the working directory to be that installation directory; then invokes mbuild.

mbuild command usage is shown below.

```
>user_dir_dir>Multics>GDixon>info>mbuild.info   (40 lines in info)
2019-07-13  mbuild, mb

Syntax as a command:  mbuild {-control_arg}...

Function: A subsystem of requests which build and prepare to install
all original-content files in the working directory (known as the
install directory).

Original-content files include: program source files, include files,
info segments, bind files, source archive(s) for a new bound segment.

The directory may contain a build script file giving details of the
installation.  This file uses the entryname of the install directory,
followed by a .mb suffix.  For example: ...>MCR10010>MCR10010.mb

For information about the Build Script Language, type:
   help build_script.gi

Control arguments:
-set_log_dir DIRNAME, -sld DIRNAME
   sets the log directory used by the update_seg initiate command.
-request STR, -rq STR
   executes STR as an mbuild request line before entering the
   request loop.
-request_loop, -rql
   enters the mbuild request loop. (Default)
-no_request_loop, -nrql
   does not enter the request.

-abbrev, -ab
   enables abbreviation expansion of request lines.
-profile path, -pf path
   specifies the pathname of the profile to use for abbreviation
   expansion.  The suffix "profile" is added if necessary.  This
   control argument implies -abbrev.
-no_abbrev, -nab
   does not enable abbreviation expansion of request lines.  (Default)

-prompt STR, -pmt STR
   sets the request loop prompt to STR.  The default is:
         ^/mbuild^[ (^d)^]:^2x
-no_prompt, -npmt
   suppresses the prompt for request lines in the request loop.
-scan, -sc
   perform a scan request of installation directory before entering
   request loop. (default)
-no_scan, -nsc
   omit scan request before entering the request loop.
```

mbuild Requests

After entering the mbuild subsystem, a subsystem prompt is displayed. To display a list of the most important mbuild requests in a typical order of use, type: `help -all`

```
mb

Installation_directory: >user_dir_dir>Multics>GDixon>work>mbuild

mbuild:  help -all
2019-07-22  Summary of mbuild Requests

list, ls                List files in the working directory (the install directory).
scan, sc                Scan the working directory for segments to build.
read, rd                Read information about build operation from a build script
                          file.

analyze, az             Analyze information gathered by scan or read requests.
progress, pg            Report progress toward completing build and install tasks.
print, p, pr            Print information about a build operation, before or after
                          analyze request.
set                     Set description or log directory; set library information for
                          segments in a build operation.
save, sv                Save information about build operation to a build script
                          file.

compile, comp           Compile source segments to create their derived Unbound_obj
                          segments.
archive_prep, arch      Update bound segment source/object archives; bind object
                          archives.
install_ec, inst        Create the Build_exec_com file which installs segments into
                          Multics Libraries.
clean, cl               Remove any derived segments found/created by prior requests.

compare, cmp            Compare original and replacement source, include, info, or
                          bindfile segments.
history, hcom           Apply history_comment command to source, include, info, or
                          bindfile segments.
lib_names, libs         Display library paths with preferred library name.
seg_type, seg           Describe the Multics Libraries policy for build/install of a
                          given segment.
paradigm, pdm           Print information about a Multics build paradigm.

library_descriptor, lds  Display information about the multics_libraries_ descriptor.
library_pathname, lpn    List pathnames in Multics Libraries matching a starname.
help, h                 Obtain detailed information on the subsystem.
.                       Identify the subsystem.
?                       Produce a list of the most commonly used requests.

mbuild:
```

To display a more complete list of requests, use the list_requests (lr) request: `lr`

To display all available requests, type: `lr -all`

The more important requests are discussed further as we introduce mbuild using several examples.

mbuild Request: help, h

The mbuild help request displays help for each of the requests supported by the mbuild subsystem. It supports fewer control arguments than the Multics help command; but provides access to help for requests internal to mbuild; for ssu standard requests used in mbuild; and for Multics commands incorporated into mbuild.

```
mbuild:  help help -all
2019-07-14  help, h

Syntax as a request:  help topic {-control_arg}...

Function: display information about a topic related to the mbuild
system or its requests.

Control arguments:
-all, -a
   displays all information available without paragraph or section prompts.
-brief, -bf
   displays a summary of request syntax, arguments, and control arguments.
-control_arg ARG_NAME, -ca ARG_NAME
   displays full information about the given argument or control argument.
```

mbuild Request: list, ls

The mbuild list request invokes the Multics list command. This permits display of segments in the installation directory without using the ssu_ command escape: .. COMMAND_LINE

NOTE: While this request invokes Multics list, remember that your list-related abbreviations are not available inside mbuild unless you use your standard [home_dir]>Person_ID.profile when invoking mbuild; or have copied your list-related abbreviations into an mbuild-specific profile.

Our first example shows an install directory MCR001 which:

- replaces an existing library segment: pnotice_language_info_.cds
- replaces an include file: cds_args.incl.pl1
- adds a new segment to the bound object: add_pnotice_supplement.pl1 to bound_pnotice_
- updates the bind file to reflect addition of the new segment: bound_pnotice_.bind

```
mb

Installation_directory: >user_dir_dir>Multics>GDixon>work>MCR001

mbuild:  ls

Segments = 4, Lengths = 7.

r w    1  bound_pnotice_.bind
r w    3  add_pnotice_supplement.pl1
r w    1  cds_args.incl.pl1
r w    3  pnotice_language_info_.cds
```

mbuild Request: scan, sc

The mbuild scan request causes mbuild to get a list of segments in the installation directory. While scanning, a Seg structure is created for each segment. This structure tries to locate an existing segment in the Multics Libraries with the same name as the segment in the install dir. If found, it assumes that library segment is being REPLACED.

Creating the Seg structure: determines the segment's library.directory and containing archive; its segment type and build paradigm. These are steps (a)-(d) in Table 8 above. For ease of reference, those steps are repeated here:

a. Scanning the directory to get a list of segments.
b. Looking in Multics Libraries to see if earlier versions are already installed in a library, thereby determining whether each segment is being added or replaced.
c. Determining whether each segment is a member of a bound segment.
d. Organizing the segments by location within the libraries, and by name.

The scan request accepts no control arguments or operands. In a clean installation directory, it prints nothing while scanning.

```
mbuild:  scan

mbuild:
```

Structure: Seg

An mbuild Seg structure begins with common substructure shared by all the mbuild structures. This common substructure holds information about a build/install operation. Subsequent elements add a set of Tlist_data structures for each list onto which the Seg may be threaded; followed by a list anchor Tlist_base structure onto which a Seg(source_arch) or Seg(object_arch) threads its component segments; followed by a set of pointers to supporting information data structures, or to other major structure types.

The following declaration (and similar declarations later in this bulletin) come from: mbuild_data_.incl.pl1.

```
dcl 1 Seg aligned based(SegP),
      2 common,                       /* Elems in: Seg, COMPILE, BOUNDOBJ, UNBOUNDOBJ, request */
        3 info,
          4 struct char(4) var,       /* = Seg, COMP, BND, or UBND                              */
          4 type char(16) var,        /* = [mbt seg Seg.name -type]                             */
          4 name char(32) var,        /* Entryname of this segment.                             */
          4 library char(32) var,     /* Library for this seg: sss, tools, hard, unb, include, */
                                      /*  or for infos: sss.info, priv.info;                    */
                                      /*  for subsystem infos: analyze_multics.info, bce.info  */
          4 operation char(12) var,   /* = "ADD", "REPLACE", "UPDATE", or "DELETE"              */
                                      /*   UPDATE designates archives whose component(s)        */
                                      /*   will be REPLACEd.  Using REPLACE for an archive      */
                                      /*   means the entire archive is REPLACEd without         */
                                      /*   changing its components during the build process.    */
          4 archive_name char(32) var,/* Name of containing archive in Multics Libraries.       */
          4 compiler char(32) var,    /* Name of compiler used to compile source segment.       */
          4 compile_options char(32) var,
                                      /* Specific compile options given in .mb file.            */
          4 produced_by_compiler char(32) var,
                                      /* Name of compiler which produced this derived segment. */
          4 pad1 char(4),
        3 request_Td aligned like Tlist_data,
                            /*             request.request_Tb -->> Seg.request_Td (Per Request) */
      2 Td,
        3 (Seg_Td,          /*              build_data.Seg_Tb -->> Seg.Seg_Td   (Per Structure) */
           pdm_Td,          /*  build_data.<paradigm-name>_Tb -->> Seg.pdm_Td    (Per Paradigm) */

           scan_Td,         /*             build_data.scan_Tb -->> Seg.scan_Td                  */
           INTERMEDIATE_Td,/*     build_data.INTERMEDIATE_Tb -->> Seg.INTERMEDIATE_Td          */

           archs_Td,        /*        BOUNDOBJ.sourceArchs_Tb -->> Seg(source_arch).archs_Td    */
                            /*        BOUNDOBJ.objectArchs_Tb -->> Seg(object_arch).archs_Td    */
           inArch_Td        /*     Seg(source_arch).inArch_Tb -->> Seg(source).inArch_Td        */
           ) aligned like Tlist_data,
      2 Tb,
        3 inArch_Tb aligned like Tlist_base,
                            /* Seg(source_arch).inArch_Tb -->> Seg(source).inArch_Td           */
      2 ptrs,
        3 seg_typeP ptr,              /* Pointer to seg_type_info structure for this seg.      */
        3 build_typeP ptr,            /* Pointer to bld_paradigm_info structure for this seg.  */

        3 name_addP ptr,              /* Pointer to list of alternate names on the seg.        */
        3 name_deleteP ptr,           /* Pointer to list of names to be deleted from the seg.  */

        3 BOUNDOBJp ptr,    /*                   Seg.BOUNDOBJp  --> BOUNDOBJ                 */
        3 COMPILEp ptr,     /*                    Seg.COMPILEp  --> COMPILE                  */
        3 UNBOUNDOBJp ptr, /*                  Seg.UNBOUNDOBJp  --> UNBOUNDOBJ               */
        3 sArchP ptr,       /*         Seg(object_arch).sArchP  --> Seg(source_arch)          */
       SegP ptr;
```

mbuild Request: print, pr, p

The mbuild print request displays information about the build/install operation. Its default output changes as scan, read, or analyze requests gather more information about the installation changeset (the segments being added, replaced, or deleted).

After the scan request for MCR001 (described above), the printed information reflects location of segments being REPLACEd; and specifies an UNKNOWN.source library for segments identified as source files being ADDed to the library.

```
mbuild:  print

Segments found by scan request:

 bound_pnotice_.archive:
   bindfile:    bound_pnotice_.bind         IN: tools.object  REPLACE;

 bound_pnotice_.s.archive:
   source:      pnotice_language_info_.cds  IN: tools.source  REPLACE compiler: cds;
 source:        add_pnotice_supplement.pl1  IN: UNKNOWN.source  ADD compiler: pl1 -ot;

 Seg(Include):       cds_args.incl.pl1      IN: sss.include   REPLACE;
```

NOTE: information displayed after a scan request is partially expressed in Build Script language, but does not obey all of the language rules. More analysis is needed to determine how these segments fit together as a changeset.

As a method for debugging mbuild, the print request can display individual threaded lists of Seg structures created during an mbuild invocation; or all of its threaded lists. The following output shows the small amount of data available for segments after a scan or read operation.

```
mbuild:  pr -list all
scan_Tb:
  Seg(bindfile):          bound_pnotice_.bind             IN: tools.object  REPLACE;
  Seg(source):            pnotice_language_info_.cds      IN: tools.source  REPLACE compiler: cds;
  Seg(source):            add_pnotice_supplement.pl1      IN: UNKNOWN.source  ADD compiler: pl1 -ot;
  Seg(Include):           cds_args.incl.pl1               IN: sss.include  REPLACE;
Seg_Tb:
  Seg(source):            pnotice_language_info_.cds      IN: tools.source  REPLACE compiler: cds;
  Seg(Include):           cds_args.incl.pl1               IN: sss.include  REPLACE;
  Seg(source):            add_pnotice_supplement.pl1      IN: UNKNOWN.source  ADD compiler: pl1 -ot;
  Seg(bindfile):          bound_pnotice_.bind             IN: tools.object  REPLACE;
```

The print request provides many types of information. Please review the help file below for more details.

```
mbuild:  help print -a
2019-07-13  print, pr, p

Syntax as a request:  pr [INFORMATION_TYPE] {-control_arg}...

Function: prints information about a build operation, before or after
analyze request.

Arguments:
INFORMATION_TYPE
   may be any of the types given in "List of Information Types".
```

```
List of Information Types:
-description, -desc
   print the current description associated with this install
   directory.  The description is specified using request: set,
   or by editing the Build Script file to add/change the Description:
   lines.
-directory, -dir
   print the current install directory pathname.
-log_dir, -ld
   print the current log directory, used in the update_seg initiate request.

-scan, -sc
   print data gathered by most recent scan of the install directory, or
   read of the build script file. (default if an analyze request has
   not been issued.)
-analyze, -az
   print installation data reshaped by an analyze request.  (default
   after an analyze request is issued.)
-new
   print original-content segments found by a scan of the install
   directory that are not mentioned in the build script.  When a
   read request loads the build script into mbuild, it reports this
   same possible omission of segments; print -new permits a repeat
   display of this data.
-unknown, -unk
   print original-content segments found by a scan of the install
   directory that have an UNKNOWN library designation.

-save_format, -save
   print installation data as it would look in the build script file.
-Seg ENTRYNAME, -seg ENTRYNAME
   print all information known about a given segment in the install
   directory.
-list {LIST_NAME}, -ls {LIST_NAME},
-thread {LIST_NAME}, -th {LIST_NAME}
   print all entries on one of the threaded lists maintained by mbuild.
   LIST_NAME may be "all" to display items on all threaded lists; or
   one of the IDs given in "List of thread selectors" below.  "all" is the
   default if no LIST_NAME is given.

Control arguments:
   Arguments to select information to be displayed for each structure.
   Structure varieties include: Seg() representing an actual segment;
   and BOUNDOBJ, UNBOUNDOBJ, and COMPILE which manage relationships
   between segments.
-all, -a
   displays all possible information for each structure type.
-structure
   displays type of structure: Seg, BOUNDOBJ, UNBOUNDOBJ, or COMPILE,
   preceding the segment's name.  If both -structure and -type are
   given, the structure and type are shown in combination:
   Seg(Include).  If neither are given, only the segment name is
   displayed.
-type, -tp
   displays the segment type based upon entryname suffix.  The
   segment's name follows this type value.  Possible values are
   returned by the request:  seg_type -type

-library, -lb
   display the target library.directory into which the segment will be
   installed.
```

```
-header, -he
   displays the operation (ADD, REPLACE, DELETE, or UPDATE); and
   compiler used to produce derived content from the source segment.
-pointer, -ptr
   displays type/name of other structures pointed to by the displayed
   structure.
-archive, -ac
   displays the target archive name in which the segment will be
   installed.  This is meaningful for source or object segments which
   are components of a bound segment archive.

-thread_base, -Tb
   displays information about threaded lists anchored in the structure.
   For example, a Seg(source_arch) anchors a list of Seg(source)
   components targeted for updating in that archive.
-thread_data, -Td
   displays information about threaded lists in which the structure is
   a member.
-name, -nm
   display names to be added to, or deleted from, the segment as it is
   installed in the library.

List of negative control arguments:
   For some types of information, extra information is displayed by
   default.  The following control arguments turn off display of
   unwanted data.
-no_structure
   stops output of structure information if it is displayed by default,
   or if -all was given earlier in request line.
-no_type, -ntp
   stops output of type information if it is displayed by default, or
   if -all was given earlier in request line.
-no_library, -nlb
   stops output of library information if it is displayed by default,
   or if -all was given earlier in request line.
-no_header,  -nhe
   stops output of header information if it is displayed by default, or
   if -all was given earlier in request line.

-no_archive, -nac
   stops output of archive information if it is displayed by default,
   or if -all was given earlier in request line.
-no_pointer, -nptr
   stops output of pointer information if it is displayed by default,
   or if -all was given earlier in request line.
-no_thread_base, -nTb
   stops output of thread anchor information if it is displayed by
   default, or if -all was given earlier in request line.
-no_thread_data, -nTd
   stops output of thread membership information if it is displayed by
   default, or if -all was given earlier in request line.
-no_name, -nnm
   stops output of names added/removed information if it is displayed
   by default, or if -all was given earlier in request line.
```

```
List of thread selectors:
all
   Display structures on all populated threads.
BOUNDOBJ, BND
   All BOUNDOBJ structures.
COMPILE, COMP
   All COMPILE structures.
UNBOUNDOBJ, UNBND
   All UNBOUNDOBJ structures.
Seg, seg
   All Seg structures.

Unbound_obj, Unb
   All Seg(Unbound_obj) structures following the Unbound_obj paradigm.
Bound_obj, Bnd
   All Seg(Bound_obj) structures following the Bound_obj paradigm.
bindfile, bind
   All Seg(bindfile) structures following the bindfile paradigm.
object_arch, oArch
   All Seg(object_arch) structures following the object_arch paradigm.
source_arch, sArch
   All Seg(source_arch) structures following the source_arch paradigm.
source, src
   All Seg(source) structures following the source paradigm.  This
   excludes sources derived from 1st step of a 2-step compile (sources
   on the INTERMEDIATE thread).
target_only, target
   All Seg(...) structures following the target_only paradigm.
listing, list
   All Seg(listing) structures following the listing paradigm.
mbuild_support, support, sup
   All Seg(Build_xxx) structures following the mbuild_support paradigm.
object_x_only, o.x
   All Seg(...) structures following the object_x_only paradigm.


scan, sc
   Original-content segments found in install dir before building.
INTERMEDIATE, inter
   Some Seg(source) structures that aren't installed because their
   content is derived from other Seg(source) files.

mbuild:
```

mbuild Request: analyze, az

The mbuild analyze request uses information about Multics Library policies and conventions, segment types and associated build paradigm's, and library location information to construct a more detailed view of segments being built and installed. It performs the following steps from Table 8 above.

e. Determining whether each segment is a source segment that must be translated or compiled into an object; that is, determining each segment's build paradigm.
f. Organizing segments by their build paradigm, to schedule the build steps appropriate to each paradigm.

After an analyze request, the print request displays a Build Script Language view of the installation.

NOTE: mbuild still does not know in which library to install the segment being ADDed. The segment is shown as an Unbound_obj, targeted for an UNKNOWN library. The intent of this example is to add the segment as a new component of bound_pnotice_. The installer must save the information below into a Build_script file, and edit that file to inform mbuild of these intentions. Those save and edit steps are described in the next sections of this bulletin.

```
mbuild:  analyze

The following segments have an UNKNOWN library:

  Unbound_obj:  add_pnotice_supplement      IN: UNKNOWN  ADD;
  source:       add_pnotice_supplement.pl1  IN: UNKNOWN.source  ADD compiler: pl1 -ot;

Please resolve these library names before doing: compile, archive_prep, or install_ec.

mbuild:  print

 Bound_obj:       bound_pnotice_                IN: tools  UPDATE;
     bindfile:    bound_pnotice_.bind            REPLACE;
     source:      pnotice_language_info_.cds     REPLACE            compiler: cds;

 Unbound_obj:     add_pnotice_supplement        IN: UNKNOWN  ADD;
     source:      add_pnotice_supplement.pl1     ADD               compiler: pl1 -ot;

 Include:         cds_args.incl.pl1             IN: sss.include  REPLACE;

mbuild:
```

IMPORTANT: After the analyze step, the installer should print and review the Build_script information to ensure that it correctly describes the intended installation; and that no UNKNOWN library references appear in the description. Incorrect analysis assumptions made by mbuild must be corrected before starting the actual build operations.

The analyze request has only one control argument.

```
mbuild:  help analyze -all
2019-10-26  analyze, az

Syntax as a request:  analyze

Function: analyzes information gathered by scan or read requests.
Knowledge of how programs and documentation are stored in libraries
like the Multics Libraries reshapes mbuild's view of segments in the
install directory.

Control arguments:
-clean, -cl
   Run the clean request if analysis completes successfully.  The
   installer is asked whether to delete the derived-content segments
   found by the analysis. (default)
-no_clean, -ncl
   Omit the clean request when analysis completes successfully.
```

Using the print request to view all threaded lists shows how the analyze request changed mbuild's view of the segments being built and installed. The enhanced views provided after analysis are discussed further in subsequent sections of this bulletin.

```
mbuild:  pr -ls all

BOUNDOBJ_Tb:
  BOUNDOBJ:               bound_pnotice_                    IN: tools  UPDATE;
                                  /^  source_archives: 1  bindfile: bound_pnotice_.bind  ^/
                                  /^  sourceArchs_Tb: 1  ^/
bindfile_Tb:
  bindfile:               bound_pnotice_.bind               IN: tools.object  REPLACE;
source_arch_Tb:
  source_arch:            bound_pnotice_.s.archive          IN: tools.source  UPDATE;
                                  /^  BOUNDOBJ: bound_pnotice_  ^/
                                  /^  inArch_Tb: 1  archs_Td  ^/
UNBOUNDOBJ_Tb:
  UNBOUNDOBJ:             add_pnotice_supplement            IN: UNKNOWN  ADD;
                                  /^  source: add_pnotice_supplement.pl1  ^/
source_Tb:
  source:                 pnotice_language_info_.cds        IN: tools.source  REPLACE compiler: cds;
                                  /^  archive: bound_pnotice_.s.archive  BOUNDOBJ: bound_pnotice_  ^/
                                  /^  inArch_Td  COMPILE ^/
  source:                 add_pnotice_supplement.pl1        IN: UNKNOWN.source  ADD compiler: pl1 -ot;
                                  /^  COMPILE  UNBOUNDOBJ  ^/
COMPILE_Tb:
  COMPILE:                pnotice_language_info_.cds        IN: tools.source;
                                  /^  source: pnotice_language_info_.cds  ^/
  COMPILE:                add_pnotice_supplement.pl1        IN: UNKNOWN.source;
                                  /^  source: add_pnotice_supplement.pl1  ^/
target_only_Tb:
  Seg(Include):           cds_args.incl.pl1                 IN: sss.include  REPLACE;
scan_Tb:
  Seg(bindfile):          bound_pnotice_.bind               IN: tools.object  REPLACE;
  Seg(source):            pnotice_language_info_.cds        IN: tools.source  REPLACE compiler: cds;
  Seg(source):            add_pnotice_supplement.pl1        IN: UNKNOWN.source  ADD compiler: pl1 -ot;
  Seg(Include):           cds_args.incl.pl1                 IN: sss.include  REPLACE;
Seg_Tb:
  Seg(source):            pnotice_language_info_.cds        IN: tools.source  REPLACE compiler: cds;
  Seg(Include):           cds_args.incl.pl1                 IN: sss.include  REPLACE;
  Seg(source):            add_pnotice_supplement.pl1        IN: UNKNOWN.source  ADD compiler: pl1 -ot;
  Seg(bindfile):          bound_pnotice_.bind               IN: tools.object  REPLACE;
  Seg(source_arch):       bound_pnotice_.s.archive          IN: tools.source  UPDATE;

mbuild:
```

mbuild Request: set

The mbuild set request supplies certain information missing from an installation directory containing only segments to be built and/or installed. The help file for this request outlines this function.

```
mbuild:  help set -all

2019-07-22  set

Syntax as a request:  set -description
                      set -log_dir DIRNAME
                      set -Seg SEG_NAME -library LIBRARY

Function: sets the installation description or log directory for an
installation; or sets the target library or library.directory for a
given segment.

After setting a new description, use the save request to capture that
description at the top of the build script file.  This permits it to be
saved/used across invocations of mbuild.

Control arguments:
-description, -desc
   prompt for a new description of the installation.  The description
   ends with a line containing only a period (.) character.
-log_dir DIRNAME, -ld DIRNAME
   sets the log directory used by the update_seg initiate command.
-Seg SEG_NAME, -seg SEG_NAME
   selects a segment to modify.
-library LIBRARY, -lb LIBRARY
   gives a new value for the Seg().library attribute.  Use one of the
   formats: <lib-name>.<dir-name>  or  <lib-name>  or  <dir-name>

List of preferred library names:
   <lib-name>.<dir-name> combinations that are short and readily
   identifiable.  Note that hard.i and mcs.i reference a directory
   named info, but it contains hardcore/communication configuration
   files and maps, not info segments.  To see directories named by
   these libraries, type: libs
```

The following example sets the description associated with the MCR001 installation effort:

```
mbuild:  set -description
Enter description (ending with a line containing only a period (.) character):
MCR001-- Test simple install of:
 - include file
 - source component for a bound object
.

mbuild:
```

mbuild Request: save, sv

The mbuild save request writes information from the analyzed view of segments being installed into a Build_script file in the installation directory. This segment has the name of the install directory, followed by a .mb suffix.

Notice use of ..print to escape a Multics print command while within mbuild. Note too that the just-created Build_script file is listed as one of the segments now in the installation directory. It has joined the list of install dir segments known to mbuild.

```
mbuild:  save

mbuild:  ..print MCR001.mb

                    MCR001.mb 07/30/19  2205.6 pdt Tue

Description:
MCR001-- Test simple install of:
 - include file
 - source component for a bound object

Installation_directory:  >user_dir_dir>Multics>GDixon>w>MCR001;

 Build_script:    MCR001.mb;

 Bound_obj:       bound_pnotice_                IN: tools  UPDATE;
     bindfile:    bound_pnotice_.bind            REPLACE;
     source:      pnotice_language_info_.cds     REPLACE            compiler: cds;

 Unbound_obj:     add_pnotice_supplement        IN: UNKNOWN  ADD;
     source:      add_pnotice_supplement.pl1     ADD                compiler: pl1 -ot;

 Include:         cds_args.incl.pl1             IN: sss.include  REPLACE;

mbuild:
```

Editing the Build_script File

After saving the Build_script into a file, the installer can edit this file to inform mbuild how to handle the new add_pnotice_supplement.pl1 source file. In this example, the intent is to add the new segment as a component of bound_pnotice_.

Edit the Build_script to remove the Unbound_obj: statement for add_pnotice_supplement. This removes the UNKNOWN target library reference. The removal also leaves the source statement for add_pnotice_supplement.pl1 positioned after the Bound_obj: major statement for bound_pnotice_. That is, the source will be ADDed as a component of the bound_pnotice_ bound segment in the tools library.

The Build_script file may be edited by escaping an ..emacs command (or the installer's favorite editor tool) from the mbuild environment. Another ..print command shows the edited Build_script file.

```
Mbuild:  ..print MCR001.mb

                    MCR001.mb 07/30/19  2219.5 pdt Tue

Description:
MCR001—Test simple install of:
 – include file
 – source component for a bound object

Installation_directory:  >user_dir_dir>Multics>Gdixon>w>MCR001;

 Build_script:    MCR001.mb;

 Bound_obj:       bound_pnotice_                IN: tools  UPDATE;
     bindfile:    bound_pnotice_.bind            REPLACE;
     source:      pnotice_language_info_.cds     REPLACE            compiler: cds;
     source:      add_pnotice_supplement.pl1     ADD                compiler: pl1 –ot;

 Include:         cds_args.incl.pl1             IN: sss.include  REPLACE;

mbuild:
```

NOTE: The installer may prefer to exit mbuild, rather than ..COMMAND escaping an editor command from with the mbuild subsystem. Both approaches reset all of mbuild's knowledge of the changeset: mbuild's internal data is released when you exit; but all data except for the list of known segments (the Seg_Tb list) is released when reading in the edited Build_script file (described in the next section of this bulletin).

When re-entering mbuild after editing the Build_script, the following requests will restore mbuild's internal data. In this case, the scan request is not needed; but it is quick and causes no problems.

```
scan; read; analyze
```

mbuild Request: read, rd

The mbuild read request parses the perhaps edited Build_script file (if one exists in the installation directory), and uses its segment information to replace what mbuild knows about the installation.

NOTE: Data from all previous scan, read, and analyze commands is replaced by new information read from the Build_script. Therefore, an analyze request is needed after the read request.

By default, the read request calls the print request to display the data just read into mbuild.

```
mbuild:  help read -all

2019-07-13  read, rd

Syntax as a request:  read {-control_arg}...

Function: reads information about the build operation from a build
script file.  If reading portions of the file containing statements
describing segments in the installation directory, that information
replaces data from any earlier scan, read, or analyze request.

Control arguments:
-all, -a
   Read the entire file.  Segment information from an earlier scan,
   read or analyze request is replaced by segment information in the
   file.  This replacement data must then be analyzed.  (default)
-description, -desc
   Read only the saved Description: at the top of the build script
   file.  Segment information obtained by earlier scan, read, and
   analyze requests is preserved.

-print, -pr
   Print segment information obtained by this read request. (default)
-brief, -bf
   Do not print segment information obtained by this read request.


mbuild:  read

Description:
MCR001-- Test simple install of:
 - include file
 - source component for a bound object

Segments found by read request:

  Build_script:    MCR001.mb;

  bound_pnotice_.archive:
    bindfile:      bound_pnotice_.bind        IN: tools.object  REPLACE;

  bound_pnotice_.s.archive:
    source:        add_pnotice_supplement.pl1 IN: tools.source  ADD compiler: pl1 -ot;
    source:        pnotice_language_info_.cds IN: tools.source  REPLACE compiler: cds;

  Seg(Include):    cds_args.incl.pl1          IN: sss.include  REPLACE;
```

Analyze after a read Request

The data parsed by a read request needs to be analyzed to gather further information. After the analyze request, mbuild's view of the segments shows the new add_pnotice_supplement.pl1 being added to bound_pnotice_.

```
mbuild:  analyze

mbuild:  p

 Build_script:     MCR001.mb;

 Bound_obj:        bound_pnotice_               IN: tools  UPDATE;
     bindfile:     bound_pnotice_.bind           REPLACE;
     source:       add_pnotice_supplement.pl1    ADD               compiler: pl1 -ot;
     source:       pnotice_language_info_.cds    REPLACE           compiler: cds;

 Include:          cds_args.incl.pl1            IN: sss.include  REPLACE;

mbuild:
```

Structure: COMPILE

The analyze request may find source segments in the install directory (segments with build paradigm = source). For such segments, analyze creates a COMPILE structure to manage the translation task. From mbuild_data_.incl.pl1:

```
  dcl 1 COMPILE aligned based(COMPILEp),
      2 common,                       /* Elems in: Seg, COMPILE, BOUNDOBJ, UNBOUNDOBJ, request  */
        3 info aligned like Seg.info,
        3 request_Td aligned like Tlist_data,
                                      /*      request.request_Tb -->> Seg.request_Td  (Per Req) */
      2 Td,
        3 COMPILE_Td aligned like Tlist_data,
                                      /*   build_data.COMPILE_Tb -->> COMPILE.COMPILE_Tb        */
      2 ptrs,
        3 sourceP ptr,                /*         COMPILE.sourceP  --> Seg(source)               */
        3 objectP ptr,                /*         COMPILE.objectP  --> Seg(Unbound_obj)          */
        3 listingP ptr,               /*        COMPILE.listingP  --> Seg(listing)              */
        3 INTERMEDIATEp ptr,          /*   COMPILE.INTERMEDIATEp  --> Seg(source)               */
                                      /*                              .produced_by_compiler^="" */
       COMPILEp ptr;
```

Each COMPILE structure points to its Seg(source) structure. After a compilation, it will point to the outputs of the compilation: a Seg(Unbound_obj) structure, and perhaps Seg(listing) and an intermediate source file Seg(source) structure (output of 1st step in a 2-step compile operation).

analyze places Seg(source) structures on a source threaded list: source_Tb; and COMPILE structures on a COMPILE threaded list: COMPILE_Tb. These lists may be displayed by a print request. Note that the list containing Seg(Unbound_obj) structures is empty (not displayed) before segments are compiled.

```
mbuild:  p -ls (source COMPILE Unbound_obj)

source_Tb:
  source:  add_pnotice_supplement.pl1        IN: tools.source  ADD compiler: pl1 -ot;
                                        /^  archive: bound_pnotice_.s.archive
                                            BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                                        /^  inArch_Td  ^/
  source:  pnotice_language_info_.cds        IN: tools.source  REPLACE compiler: cds;
                                        /^  archive: bound_pnotice_.s.archive
                                            BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                                        /^  inArch_Td  ^/
COMPILE_Tb:
  COMPILE: add_pnotice_supplement.pl1        IN: tools.source;
                                        /^  source: add_pnotice_supplement.pl1  ^/
  COMPILE: pnotice_language_info_.cds        IN: tools.source;
                                        /^  source: pnotice_language_info_.cds  ^/
```

mbuild Request: progress, pg

The mbuild progress request shows completion status for the requests involved in a normal build/install effort. progress information is obtained by asking the requests supervising the build effort two questions:

- Has mbuild provided sufficient data to perform this request?
- If so, have tasks associated with the request been completed for all segments? Are their expected outputs actually present in the install directory?

If the answer to either question is no, then the request is not complete.

The following output shows progress for MCR001 after the above analyze request.

```
mbuild:  progress

   COMPLETED   REQUESTS for Builds
   _________   ___________________________
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
               compile
               archive_prep
               install_ec
```

The progress indicates that analysis is complete, and that segments in the install directory need to be compiled. Therefore, compile should be the next request in the build effort.

mbuild Request: compile, comp

The mbuild compile request accomplishes the next step in Table 8.

> g. Compiling segments that require translation, and tracking the success and outputs of each compilation.

Compile options specified in each source statement are used when compiling that segment. Any options given with the compile request are added to those in the source statement.

```
mbuild:  help compile -all
2019-07-13  compile, comp

Syntax as a request:  comp {SEG_NAME} {-control_arg}...

Function: compiles source segments to create their derived Unbound_obj
segments.

Arguments:
SEG_NAME
   names a single segment to compile.  The name includes the language
   suffix (e.g., source.pl1).  If not given, all segments in the
   COMPILE list are compiled.  Useful to recompile a single source
   after correcting a compilation error.

Control arguments:
   If given, the follow control arguments are added to any compile
   option given in the source statement for each segment.
-list, -ls
   for a PL/I compilation, produces a source program listing followed
   by a list of all the names used in the compilation, followed by an
   assembly-like listing of the compiled object program.  Listing
   produced by other compilers may have other content.
-table, -tb
   for a PL/I compilation, produces an Unbound_obj output containing a
   full symbol table for use by probe and debug commands.
```

For MCR001, the compile request translates both source segments. The actual command invoking the compiler for each source segment is shown, followed by any messages from that compilation.

```
mbuild:  compile

---------- pl1 add_pnotice_supplement.pl1 -ot

PL/1 33f

---------- cds pnotice_language_info_.cds

CDS - PL/1 33f

mbuild:
```

Each compilation produces an executable object segment. Since the compile does not associate this object as a component of any bound segment, mbuild tracks it as an Unbound_obj which is linked to its COMPILE structure. In the output below, both source and Unbound_obj are tied to their COMPILE structure.

```
mbuild:  p -ls (source COMPILE Unbound_obj)
source_Tb:
  source:  add_pnotice_supplement.pl1        IN: tools.source  ADD compiler: pl1 -ot;
                                        /^  archive: bound_pnotice_.s.archive
                                            BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                                        /^  inArch_Td  ^/
  source:  pnotice_language_info_.cds        IN: tools.source  REPLACE compiler: cds;
                                        /^  archive: bound_pnotice_.s.archive
                                            BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                                        /^  inArch_Td  ^/
COMPILE_Tb:
  COMPILE: add_pnotice_supplement.pl1        IN: tools.source;
                                        /^  source: add_pnotice_supplement.pl1
                                            object: add_pnotice_supplement  ^/
  COMPILE: pnotice_language_info_.cds        IN: tools.source;
                                        /^  source: pnotice_language_info_.cds
                                            object: pnotice_language_info_  ^/
Unbound_obj_Tb:
  Unbound_obj:  add_pnotice_supplement       IN: tools.object  ADD;
                                        /^  COMPILE  ^/
  Unbound_obj:  pnotice_language_info_       IN: tools.object  REPLACE;
                                        /^  archive: bound_pnotice_.archive COMPILE ^/


mbuild:  progress

   COMPLETED   REQUESTS for Builds
   ---------   ---------------------------
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
      yes      compile
               archive_prep
               install_ec
```

Structure: BOUNDOBJ

The analyze request may find references to bound segments in the installation directory. The directory may hold a bound_XXX_.bind file; or contain source segments which are found in a source archive by a library search; or it may contain a Build_script file including a Bound_obj statement. analyze uses these clues to fabricate a BOUNDOBJ structure for each bound segment being updated.

This BOUNDOBJ structure contains:

- a list of source_arch structures representing the archive(s) holding source components being ADDed or REPLACEd;
- a pointer to any bindfile structure for a .bind file being REPLACEd or ADDed;
- before binding, will hold a list of object archives to be updated; for a multi-archive bound object, this list sometimes holds references to other object archives which are needed to re-bind the bound object, even though they are not updated by unbound object segments compiled from updated sources;
- a pointer to a segment structure identifying the bound object produced by the bind command; and
- an optional bind map or bind listing describing the binding operation and resultant bound object segment.

```
dcl 1 BOUNDOBJ aligned based(BOUNDOBJp),
    2 common,           /* Elems in: Seg, COMPILE, BOUNDOBJ, UNBOUNDOBJ, request  */
      3 info aligned like Seg.info,
      3 request_Td aligned like Tlist_data,
                        /*      request.request_Tb -->> Seg.request_Td  (Per Req) */
    2 Td,
      3 BOUNDOBJ_Td aligned like Tlist_data,
                        /*  build_data.BOUNDOBJ_Tb -->> BOUNDOBJ.BOUNDOBJ_Tb      */
    2 Tb,
      3 (sourceArchs_Tb,/* BOUNDOBJ.sourceArchs_Tb -->> Seg(source_arch).archs_Td */
         objectArchs_Tb /* BOUNDOBJ.objectArchs_Tb -->> Seg(object_arch).archs_Td */
         ) aligned like Tlist_base,
    2 ptrs,
      3 (bindfileP,     /*      BOUNDOBJ.bindfileP  --> Seg(bindfile)             */
         Bound_objP,    /*     BOUNDOBJ.bound_objP  --> Seg(Bound_obj)            */
         listingP       /*       BOUNDOBJ.listingP  --> Seg(listing)              */
         ) ptr,
    2 sourceArchsN fixed bin,
                        /* Number of source archives for bound seg in libraries.  */
     BOUNDOBJp ptr;
```

Structures associated with a BOUNDOBJ by analysis of the installation directory are shown below.

```
mbuild:  pr -ls (BOUNDOBJ bindfile source_arch source)
BOUNDOBJ_Tb:
  BOUNDOBJ:      bound_pnotice_               IN: tools  UPDATE;
                             /^  source_archives: 1  bindfile: bound_pnotice_.bind  ^/
                             /^  sourceArchs_Tb: 1  ^/
bindfile_Tb:
  bindfile:      bound_pnotice_.bind          IN: tools.object  REPLACE;
source_arch_Tb:
  source_arch:   bound_pnotice_.s.archive     IN: tools.source  UPDATE;
                             /^  BOUNDOBJ: bound_pnotice_  ^/
                             /^  inArch_Tb: 2  archs_Td  ^/
source_Tb:
  source:        add_pnotice_supplement.pl1   IN: tools.source  ADD compiler: pl1 -ot;
                             /^  archive: bound_pnotice_.s.archive
                                 BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                             /^  inArch_Td  ^/
  source:        pnotice_language_info_.cds   IN: tools.source  REPLACE compiler: cds;
                             /^  archive: bound_pnotice_.s.archive
                                 BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                             /^  inArch_Td  ^/
```

mbuild Request: archive_prep, arch

The mbuild archive_prep request performs tasks which group individual unbound object segments into a single bound object segment. This includes the following tasks from Table 8.

h. Grouping members of a bound segment into source archives, and object archives.
i. Binding components of a bound object archive into the bound object segment.

The archive_prep request accepts only a few arguments to request generating a listing describing how component objects are grouped into the bound object segment.

```
mbuild:  help archive_prep -all
2019-07-13  archive_prep, arch

Syntax as a request:  arch {-control_arg}

Function: updates bound segment source/object archives.  Bind object
archives to produce a bound segment.

Control arguments:
-list, -ls
   produces a listing segment whose name is derived from the name of
   the bound object segment plus a suffix of list.  The listing segment
   is generated to dprint; it contains the bound segment's bind control
   segment (see "Notes on bindfile"), its bind map, and that
   information from the bound object segment printed by the
   print_link_info command.  You can't invoke -list with -map.  In the
   absence of -list or -map, no listing segment is generated.
-map
   produces a listing segment (with the suffixes list and map) that
   contains only the bind map information.  It is incompatible with
   -list.  In the absence of -list or -map, no listing segment is
   generated.
```

Tasks performed by the archive_prep request include:

- Fetch a copy of each source archive being UPDATEd by the installation.
- Add/update new/existing source segments into their targeted source archive.
- Fetch a copy of each object archive corresponding to the updated source archive(s).
- Add/update new/existing object segments derived from source files into that corresponding object archive.
- Add/update any new/replaced bindfile into the first object archive.
- Bind updated object archive(s) (including any unmodified object archives from the library), to produce a modified Bound_obj segment.

Output from the request for our MCR001 example (one bound segment being updated, composed of 1 source archive and its corresponding object archive) is shown below. Each source file remains in the installation directory while a copy is added or updated into the source archive. Derived Unbound_obj segments are added/updated to the object archive, then deleted from the installation directory.

```
mbuild:  archive_prep

---------- library_fetch bound_pnotice_.s.archive -lb tools.source

   archive a bound_pnotice_.s.archive add_pnotice_supplement.pl1

   archive u bound_pnotice_.s.archive pnotice_language_info_.cds

---------- library_fetch bound_pnotice_.archive -lb tools.object

   archive u bound_pnotice_.archive bound_pnotice_.bind

   archive adf bound_pnotice_.archive add_pnotice_supplement

   archive udf bound_pnotice_.archive pnotice_language_info_

----------  bind  bound_pnotice_.archive

Binding bound_pnotice_
```

The archive_prep request updates the BOUNDOBJ structure by populating its object_arch list with Seg(object_arch) structure(s); and pointing to the Seg(Bound_obj) generated by the bind command.

```
mbuild:  p -ls (BOUNDOBJ bindfile Bound_obj source_arch object_arch source)
BOUNDOBJ_Tb:
  BOUNDOBJ:            bound_pnotice_                    IN: tools  UPDATE;
                         /^  source_archives: 1  bindfile: bound_pnotice_.bind  Bound_obj  ^/
                         /^  sourceArchs_Tb: 1  objectArchs_Tb: 1  ^/
bindfile_Tb:
  bindfile:            bound_pnotice_.bind               IN: tools.object  REPLACE;
Bound_obj_Tb:
  Bound_obj:           bound_pnotice_                    IN: tools.execution  REPLACE;
source_arch_Tb:
  source_arch:         bound_pnotice_.s.archive          IN: tools.source  UPDATE;
                         /^  BOUNDOBJ: bound_pnotice_  ^/
                         /^  inArch_Tb: 2  archs_Td  ^/
object_arch_Tb:
  object_arch:         bound_pnotice_.archive            IN: tools.object  UPDATE compiler: bind;
                         /^  BOUNDOBJ: bound_pnotice_  sArch: bound_pnotice_.s.archive  ^/
                         /^  archs_Td  ^/
source_Tb:
  source:              add_pnotice_supplement.pl1        IN: tools.source  ADD compiler: pl1 -ot;
                         /^  archive: bound_pnotice_.s.archive
                             BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                         /^  inArch_Td  ^/
  source:              pnotice_language_info_.cds        IN: tools.source  REPLACE compiler: cds;
                         /^  archive: bound_pnotice_.s.archive
                             BOUNDOBJ: bound_pnotice_  COMPILE  ^/
                         /^  inArch_Td  ^/
```

The progress request now shows all necessary requests completed, except for the install_ec request.

```
mbuild:  pg

   COMPLETED   REQUESTS for Builds
   ---------   ----------------------------
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
      yes      compile
      yes      archive_prep
               install_ec
```

mbuild Request: install_ec, inst

The mbuild install_ec request sets up the update_seg command to install the changeset. It implements the following task from Table 8.

> j. Issuing commands to install both original-content segments and their derived-content objects into the Multics Libraries.

update_seg performs an excellent job of safely adding, replacing or deleting individual segments within the Multics Libraries while the system is in operation and existing library segments may be in-use. The difficulty in using this tool arises because installer must know which segments to install; where to install them; what ACLs and ring brackets should be placed on the segments; what names to place on installed segments; etc. In short, the installer must understand the install paradigm for each segment type to be installed; and in which library the segment should be installed.

mbuild knows the install paradigm for each segment type; it knows the target library. Its install_ec request can use segment information in the mbuild data to issue the correct update_seg commands.

However, there may well be unusual installation cases not covered completely or correctly by mbuild's install paradigms. Cases such as installing new inner-ring gate segments, or inner-ring database segments may require special access or naming operations unknown to mbuild. Therefore, the installer must be given an opportunity to tweak, or tailor, or even augment the update_seg instructions fabricated by mbuild.

The mbuild install_ec request enables this installer oversight by writing its setup instructions for update_seg into a Build_exec_com file (e.g., MCR001.mb.ec). Comments within the file describe each type of segment being installed, and identify target library.directory using an absolute library pathname. Commands in the exec_com will:

- Create an update_seg installation object segment (database for subsequent us commands) called a Build_io segment (e.g., MCR001.mb.io). The mbuild description field and log directory options (if set) are included in this setup information.
- Define a separate section of the exec_com for each type of segment being installed (executable objects, source and object archives and standalones; include files; info segments; listings; and other types of segment).
    - Begin each section with an update_seg initiate command, setting appropriate ACL and ring brackets; and commenting on the type of segment installed by that section.
    - Add update_seg add, replace, and delete commands which install segments of the given type.
- After all sections are handled, it will end the exec_com with an update_seg print and list command, giving the installer who eventually runs the Build_exec_com update_seg's interpretation of the commands; and allowing update_seg to report any errors it encounters in preparing the installation.

The install_ec request has only one control argument, specifying whether to install listings generated by compile or bind operations.

```
mbuild:  help install_ec -all
2019-07-21  install_ec, inst

Syntax as a request:  inst {-control_arg}

Function: creates the Build_exec_com file, containing update_seg
commands that install the segments identified in the Build_script.

Control arguments:
-list, -ls
   installs any .list segments found in the installation directory
   into the LIBRARY_NAME.listings target directory.
```

The install_ec request decides which segments to install using the segment database constructed by all the preceding commands in the current mbuild invocation: scan and/or read; analyze; compile; archive_prep.

- For executable segments, it walks down the BOUNDOBJ, UNBOUNDOBJ, and object_x_only lists, selecting the executable segments associated with each structure.

- For source and object, it walks down these same BOUNDOBJ, UNBOUNDOBJ and object_x_only lists, selecting: the updated or added source_arch and object_arch components of a BOUNDOBJ; the source and object segments of an UNBOUND_OBJ; and the segment on the object_x_only list.

- For include files, it walks down the target_only list, looking for Seg(Include) segments.

- For info segments, it walks down the target_only list, looking for Seg(Info) segments.

- For other types of segment, it walks down the target_only list, looking for Seg() structures which are not of type Include or Info.

- If listings are being installed, it walks down the listings list, installing each segment on that list.

The install_ec request does not display any output when executed. It uses a vfile_ attachment to fabricate the Build_exec_com file. If the request is rerun, any existing Build_exec_com file is truncated before update_seg commands are written to the file.

The following shows generating a Build_exec_com to install segments of our MCR001 example.

```
mbuild:  set -log_dir >sysbuild>MCRs

mbuild:  install_ec

mbuild:  ..pr MCR001.mb.ec

                    MCR001.mb.ec        09/20/19  0810.0 pdt Fri


&version 2
&trace &command off
&-
delete MCR001.mb.i? -brief -query_all
us sd                   -acl re *.*.*  -rb 1 5 5     &- ---- set global defaults ----
&-
&attach
us in MCR001.mb.io -set_log_dir >sysbuild>MCRs -log -no_fill
MCR001-- Test simple install of:
 - include file
 - source component for a bound object
.
&detach
&-
us in                   -acl re *.*.*               &- -------- executable ---------
us rp  bound_pnotice_                   >tools>== -ss
&-
us in                   -acl r  *.*.*               &- -------- source, object -----
us rp  bound_pnotice_.s.archive         >ldd>tools>source>== -ac
us rp  bound_pnotice_.archive           >ldd>tools>object>== -ac
&-
us in                   -acl r  *.*.*               &- -------- Include ------------
us rp  cds_args.incl.pl1                >ldd>include>==
&-
us ls
us pr

mbuild:
```

When the Build_exec_com is run, it will produce the Build_io segment (e.g., MCR001.mb.io) which contains update_seg's database; and a Build_log segment (e.g., MCR001.mb.il) which describes the detailed steps taken by update_seg to implement each us add, replace (rp) or delete (dl) command.

IMPORTANT: The installer should carefully review both the Build_exec_com and Build_log segments to ensure the correct list of segments are being installed into the proper target library directories, with necessary names on each segment, and correct ACL and ring brackets on each target segment. These details are spelled out in the Build_log. If all details are correct, the installer uses the following command to perform the installation:

update_seg install

For details about the update_seg command, refer to:

AN80: Multics Library Maintenance Program Logic Manual

mbuild Request: clean, cl

The mbuild clean request performs the final step listed in Table 8.

> k. Cleanup after the installation (to remove derived-content objects, and reduce storage space used by the installation directory). This could involve deleting: the entire directory; or remnants of the bind or install parts; or all derived-content segments, etc.

Options allow the installer to selectively remove certain segments, based upon the build/install progress.

```
mbuild:  help clean -all
2019-07-24  clean, cl

Syntax as a request:  cl {-control_arg}...

Function: Removes derived segments found/created by prior mbuild
requests, or by the update_seg install command.  Use a control argument
to reduce or widen the range of files removed.

Control arguments:
-all, -a
   Removes all files created by any mbuild request, except the
   Build_script file (created by an earlier save request).  Used when
   starting over on a build effort when compile or archive_prep
   requests report errors; or after: update_seg de_install
-installed, -inst
   Removes derived-content segment created by a build effort, leaving
   only: original-content segments; Build_script; Build_exec_com; and
   segments created by running the update_seg install command: the
   Build_io (.mb.io) installation object file; and the Build_log
   (.mb.il).  (default if neither -all nor -list is given.)
-listings, -list, -ls
   Removes only listings (.list segments) created by the build effort.
-intermediate, -inter
   Removes only intermediate source segments (those created by the 1st
   step of a 2-step translation.

-query_all, -qya
   Lists segments to be removed and asks queries whether they should be
   deleted or not.  (default)
-no_query_all, -nqya
   Removes segments without a query.  Segments removed are still
   listed.
```

By default, the request displays a list of segments matching the cleanup criteria, and asks for permission to remove those segments.

```
mbuild:  clean
  Derived-content segments eligible for clean:
     Bound_obj:     bound_pnotice_
     source_arch:   bound_pnotice_.s.archive
     object_arch:   bound_pnotice_.archive

mbuild (clean): Delete the segments above?   no

mbuild:  clean -all
  Derived-content segments eligible for clean:
     Bound_obj:     bound_pnotice_
     source_arch:   bound_pnotice_.s.archive
     object_arch:   bound_pnotice_.archive
     Build_exec_com: MCR001.mb.ec

mbuild (clean): Delete the segments above?   y

mbuild:  ls

Segments = 5, Lengths = 9.

r w    1  bound_pnotice_.bind
r w    3  add_pnotice_supplement.pl1
r w    1  MCR001.mb
r w    1  cds_args.incl.pl1
r w    3  pnotice_language_info_.cds
```

mbuild Request: lib_names, libs

The mbuild lib_names request displays the preferred library.directory names to use as a value in the IN: clause of statements in Build Script Language; or as values in the request:
set -seg SEG_NAME -library LIBRARY.DIRECTORY

These names come from the library descriptor database describing the Multics Libraries; or from an alternate library descriptor describing a private library. For information about private libraries, see the section of this bulletin called: Using mbuild with a Private Library.

```
mbuild:  help libs -all
2019-07-22  lib_names, libs

Syntax as a request:  libs {-control_arg}

Function: Display library paths with preferred library name.

Control arguments:
-library, -lb
   Display library root directories defined in the current library
   descriptor, and the preferred library name identifying each
   directory. (default)
-directory, -dir
   Display the preferred <dir-name> components for 2-component library
   names <lib-name>.<dir-name> used by mbuild.
-analyze, -az, -analysis, -anal
   Show how preferred library names were devised for each root
   directory of the current library descriptor.
```

An excerpt from the default output of this request is shown below.

```
mbuild:  lib_names
  PREFERRED LIBRARY     ROOT PATHNAME                  COMMENT
  -------------------   ----------------------------- ----------
       sss.source        >ldd>sss>source
       sss.object        >ldd>sss>object
      sss.listings       >ldd>listings>sss
     sss.execution       >sss
       unb.source        >ldd>unb>source
       unb.object        >ldd>unb>object
      unb.listings       >ldd>listings>unbundled_1      library identifies several lib root dirs.
      unb.listings       >ldd>listings>unbundled_2      library identifies several lib root dirs.
     unb.execution       >unb
      tools.source       >ldd>tools>source
      tools.object       >ldd>tools>object
     tools.listings      >ldd>listings>tools
    tools.execution      >tools
      hard.source        >ldd>hard>source
     hard.execution      >ldd>hard>execution
      hard.object        >ldd>hard>object
     hard.listings       >ldd>listings>hard
         hard.i          >ldd>hard>info
       mcs.source        >ldd>mcs>source
       mcs.object        >ldd>mcs>object
         mcs.i           >ldd>mcs>info
      mcs.listings       >ldd>listings>comm
       obs.source        >ldd>obs>source
       obs.object        >ldd>obs>object
     obs.execution       >obs
      sss.include        >ldd>include
        sss.info         >doc>info
       priv.info         >doc>privileged
```

Additional mbuild Requests

Two tools mentioned in Table 2 relate to programming and auditing:

- compare_ascii, cpa: mbuild knows the location of library versions of segments being replaced. mbuild could provide a request to run compare_ascii on some (or all) of these replaced segments.

- history_comments: mbuild could be trained about which segment types allow history comments within their contents, to track content changes. It could provide a request to:
  - display incomplete history comments (reminding the developer to add a new history comment);
  - update comments with MCR number;
  - update comments with auditor approval;
  - update comments with installer date and target release information.

mbuild Request: compare, cmp

The mbuild compare request executes a compare_ascii command against some or all of the original-content source, include, info, and bindfile segments in the install directory.

```
2019-09-09  compare, cmp

Syntax as a request:  cmp {SEG_NAME} {-cpa_control_arg}...

Arguments:
SEG_NAME
   names the segment(s) to compare.  The name includes the language
   suffix (e.g., source.pl1).  If not given, all source, include,
   info, and bindfile segments being REPLACEd are compared.  The
   star convention may be used to select several segments.

Control arguments:
   Any control argument accepted by compare_ascii (cpa) command may
   be given, except -original and -no_original.
```

mbuild Request: history, hcom

The mbuild hcom request executes a history_comment command or active request against some or all of the original-content source, include, info and bindfile segments in the install directory.

```
2019-09-09  history, hcom

Syntax as a request:
   hcom HCOM_OPERATION {SEG_NAME} {HCOM_ARGS_AND_CONTROL_ARGS}...

Syntax as an active request:
   [hcom HCOM_OPERATION {SEG_NAME} {HCOM_ARGS_AND_CONTROL_ARGS}...]

Arguments:
HCOM_OPERATION
   names the history_comment operation to perform on each segment.  See
   "List of operations" below.
SEG_NAME
   names the original-content segment(s) to process.  The name includes
   the language suffix (e.g., probe.pl1).  If not given, all
   original-content source, include, info, and bindfile segments in the
   installation directory are processed.  The star convention may be
   used to select several segments.
HCOM_ARGS_AND_CONTROL_ARGS
   Arguments and control arguments accepted by the
   "history_comment HCOM_OPERATION" command may be given.  For argument
   details, type:
     help hcom.HCOM_OPERATION
   where HCOM_OPERATION is an operation from "List of operations" below.
   Arguments containing special characters must be double-quoted.

List of operations:
add
   adds a new history comment to a source program.
add_field, af
   add missing fields to an existing history comment.
replace_field, rpf
   replaces fields in an existing history comment.
display, ds
   displays one or more history comments in a source program.
format, fmt
   reformats history comments in a source program, placing them in
   standard form.

check, ck
   checks history comments in a source program prior to its submission
   for installation to ensure that all fields except the INSTALL_ID are
   present.
compare, cmp
   displays the differences, if any, between the source and original
   modules.
exists
   checks to see if a history comment with certain attributes exists.
get
   gets one or more field values from selected history comments.
```

```
install
   checks history comments in a source program prior to its
   installation for completeness, and updates the INSTALL_ID field.

History comment format:  Following is a pl1 history comment example.
Other languages will have the comment delimiters appropriate for their
respective language.

 /****^  HISTORY COMMENTS:
 1)  change(1985-05-12, HSmith), approve(1985-05-25, MCR2355),
     audit(1985-05-26, Jones), install(1985-05-30, MR11.0-3382):
     Increased size of test_array to eliminate subscript error.
 2)  change(1985-05-28, HSmith), approve(1985-05-29, MCR2356),
     audit(1985-06-05, Rogers), install(1985-06-10, MR11.0-3384):
     Added the -brief and -long control arguments.
                                          END HISTORY COMMENTS */

Notes:  To determine if prior history comments exist in the module,
the source module is checked for a line containing the history
comment block beginning; i.e., a line beginning with the appropriate
comment delimiter, containing the word "HISTORY", and containing the
word "COMMENTS:".  If this is found, the program then checks for the
history comment block ending; i.e., a line containing "END HISTORY
COMMENTS".

List of history comment fields:
   The fields within a given history comment are identified as follows:

   NO) change  (CHANGE_DATE,  CHANGE_PERSON_ID),
       approve (APPROVE_DATE, APPROVE_ID),
       audit   (AUDIT_DATE,   AUDITOR_PERSON_ID),
       install (INSTALL_DATE, INSTALL_ID):  SUMMARY

   The  fields  in a history comment are named as described below.  The
   sample  validation  routine, hcom_default_validate_, validates field
   formats  used  by the Multics Development Center as described below.
   However,  each  site  may provide its own validation routines to
   tailor the contents of the user-settable field values.

NO
   is  the  number  of  the  history  comment.   Comments  are numbered
   sequentially in chronological order,  starting with 1.  (supplied by
   hcom)
CHANGE_DATE
   date  (yyyy-mm-dd) on  which history  comment was  first added  to the
   source module.  (supplied by hcom)
CHANGE_PERSON_ID
   person_id of the person who added the history comment.  (supplied by
   hcom)
APPROVE_DATE
   date  (yyyy-mm-dd)  on  which  an  approval  value  was supplied for a
   history comment.  (supplied by hcom)
```

APPROVE_ID
identifier authorizing the change. The default validation routine expects an identifier in the form "TYPEnnnn" for MCRs (Multics Change Request), PBFs (Post-installation Bug Fix associated with MCRnnnn) or MECRs (Multics Emergency Change Request); i.e., MCR6734, PBF6734, MECR0102. For critical fixes the identifier should be in the form of fix_nnnn or fix_nnnn.ds. The maximum length of this field is 24 characters. (supplied by user)
AUDIT_DATE
date (yyyy-mm-dd) audit field added to history comment. (supplied by hcom)
AUDIT_PERSON_ID
person_id of the person who audited the source module. (supplied by hcom)

INSTALL_DATE
date (yyyy-mm-dd) install field added to history comment. (supplied by hcom)
INSTALL_ID
value identifying either a specific installation or the installer of a critical fix. The default validation routine expects an identifier in the form "MRrel-nnnnn", consisting of a release number and installation sequence counter, e.g., MR12.0-00234. For a critical fix, the validation routine expects a person-id naming the person who installed the fix. The maximum length of this field is 24 characters. (supplied by user)
SUMMARY
brief description of the change made to the module. This field contains text (up to 2000 characters) and is not validated. (supplied by user)

Notes: The following is a typical usage pattern expected for the various operations of the history_comment command.

o The developer makes a change to the source module. He could add a new history comment by hand (perhaps using an Emacs extension to prompt for field values). Or after adding the change, he could use the history_comment add operation to add a new comment. A typical command line might be:

```
hcom add prog.pl1
```

o The developer may not have had approval for the change at the time the history comment was added. When approval is gained, he can use the history_comment add_field operation to add the approve field. For example:

```
hcom af prog.pl1 -approve MCR7235
```

o The developer can display the history comments in a program, or even compare the comments in a modified version of a program with those in the library copy of the program. For example:

```
hcom display prog.pl1 new
```

would display the new history comments in the source module, while

```
hcom compare prog.pl1
```

would display the differences between the source module and the original module.

o When the change is audited, the auditor uses the history_comment add_field operation to supply an audit field for all new or incomplete history comments. For example:

```
hcom af prog.pl1 -audit
```

o When the developer is ready to submit the change for installation, he uses the history_comment check operation to ensure that all comment fields except the install field have been supplied in each changed module. Since the developer has a site-defined validation routine called hcom_site_validate_ in his object search rules, this routine is used to fully validate the fields of all comments.

```
hcom check prog.pl1
```

o When the installer receives the modules in an installation, he uses the history_comment install operation to ensure that new history comments describing the changes are present. The install operation also adds an identifier to each new comment, indicating in which installation it was installed. The installer can use a special library-defined validtion routine to perform special field validations. In the example below, this library validation routine is called hcom_mdc_validate_.

```
hcom install prog.pl1 -vdt hcom_mdc_validate_ -install MR12.0-0023
```

List of Related Info Segments:
Additional information may be obtained on the history comment operations by referring to the following info segments:

history_comment.add.info (hcom.add.info)
history_comment.add_field.info (hcom.af.info)
history_comment.check.info (hcom.ck.info)
history_comment.compare.info (hcom.cmp.info)
history_comment.display.info (hcom.ds.info)
history_comment.exists.info (hcom.exists.info)
history_comment.format.info (hcom.fmt.info)
history_comment.get.info (hcom.get.info)
history_comment.install.info (hcom.install.info)
history_comment.replace_field.info (hcom.rpf.info)

Other mbuild Examples

Earlier sections of this bulletin used a typical changeset to introduce mbuild requests; a changeset that replaced a component of an existing bound segment, added a new component, and replaced an include file.

Other changesets might deal with adding or updating unbound segments; adding an entire new bound segment; replacing non-executable segments (include files or info segments); and deleting library segments, or components of a bound segment. The next few subsections quickly describe how these changesets are handled.

Add or Replace Unbound Objects

A changeset dealing with unbound objects starts with an installation directory containing source files for the unbound objects being ADDed or REPLACEd. In this example, two gate objects and a data archive are being installed.

```
ls

Segments = 4, Lengths = 12.

r w    1  MCR003.mb
rew    4  new_gate_.alm
r w    4  hcs_.alm
r w    3  asu_data.archive
```

After using scan and analyze requests, mbuild identified two segments being REPLACEd, and assumes one unknown segment is being ADDed.

```
mbuild:  scan

mbuild:  az

The following segments have an UNKNOWN library:

  Unbound_obj:             new_gate_            IN: UNKNOWN  ADD;
  source:                  new_gate_.alm        IN: UNKNOWN.source  ADD compiler: alm;

Please resolve these library names before doing: compile, archive_prep, or install_ec.

mbuild:  pr

 Build_script:            MCR003.mb;

 Unbound_obj:             hcs_                     IN: hard  REPLACE;
     source:              hcs_.alm                  REPLACE           compiler: alm;

 Unbound_obj:             new_gate_                IN: UNKNOWN  ADD;
     source:              new_gate_.alm             ADD               compiler: alm;

 Seg(data_arch):          asu_data.archive         IN: tools.execution  REPLACE;
```

The edited Build_script file (creation and editing not shown here) can be read to supply missing information about the segment being added, including specifying additional names for its Unbound_obj.

```
mbuild:  rd

Description:
Test unusual cases:
 - Replacing an unbound segment (a gate).
 - Adding an unbound segment (a new gate).
 - Replacing an archive which is not a component of a bound seg (data_arch).
 - Adding names to an unbound segment (edit build script manually to include the
names).
```

```
Segments found by read request:

  Build_script:            MCR003.mb;

  source:                  hcs_.alm            IN: hard.source  REPLACE compiler: alm;
  Unbound_obj:             new_gate_           IN: tools  ADD;
                             add_name:
                               new_priv_gate_
                               new_admin_gate_;
  source:                  new_gate_.alm       IN: tools.source  ADD compiler: alm;

  Seg(data_arch):          asu_data.archive    IN: tools.execution  REPLACE;


mbuild:  analyze; print

 Build_script:             MCR003.mb;

 Unbound_obj:              new_gate_           IN: tools  ADD;
                             add_name:
                               new_priv_gate_
                               new_admin_gate_;
     source:               new_gate_.alm        ADD                compiler: alm;

 Unbound_obj:              hcs_                IN: hard  REPLACE;
     source:               hcs_.alm             REPLACE            compiler: alm;

 Seg(data_arch):           asu_data.archive    IN: tools.execution  REPLACE;
```

The progress request indicates the changeset is ready for the compile request. The two gate source segments are compiled. In addition, added names given in the Build_script are applied to one of the gate objects.

```
mbuild:  pg

   COMPLETED   REQUESTS for Builds
   ---------   ---------------------------
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
               compile
               archive_prep
               install_ec

mbuild:  compile

---------- alm hcs_.alm

ALM 8.14

---------- alm new_gate_.alm

ALM 8.14
add_name new_gate_  new_priv_gate_ new_admin_gate_ -bf
```

After compiling, a list of the installation directory shows the compiler outputs: two gate objects. A progress request shows there are no segments to archive or bind. The changeset is ready to install.

```
mbuild:  ls

Segments = 6, Lengths = 28.

re     8  new_gate_
          new_priv_gate_
          new_admin_gate_
re     8  hcs_
r w    1  MCR003.mb
rew    4  new_gate_.alm
r w    4  hcs_.alm
r w    3  asu_data.archive

mbuild:  pg

   COMPLETED   REQUESTS for Builds
   _________   ____________________________
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
      yes      compile
      yes      archive_prep
               install_ec


mbuild:  install_ec
```

```
mbuild:  ..pr MCR003.mb.ec

                    MCR003.mb.ec        09/20/19  0817.7 pdt Fri


&version 2
&trace &command off
&-
delete MCR003.mb.i? -brief -query_all
us sd                    -acl re *.*.*  -rb 1 5 5     &- ---- set global defaults ----
&-
&attach
us in MCR003.mb.io -log -no_fill
Test unusual cases:
 - Replacing an unbound segment (a gate).
 - Adding an unbound segment (a new gate).
 - Replacing an archive which is not a component of a bound seg (data_arch).
 - Adding names to an unbound segment (edit build script manually to include the
names).
.
&detach
&-
us in                    -acl re *.*.*               &- -------- executable ---------
us add new_gate_                          >tools>== -ss
us rp  hcs_                               >ldd>hard>execution>== -ss
&-
us in                    -acl r  *.*.*               &- -------- source, object -----
us add new_gate_.alm                      >ldd>tools>source>==
us add new_gate_                          >ldd>tools>object>==
us rp  hcs_                               >ldd>hard>object>==
&-
us ls
us pr
```

The update_seg command which adds the new_gate_ segment into >tools is INCORRECT. It uses the default ring brackets of 1,5,5, which are appropriate for most executable segments being installed, but not for gate segments. This new_gate_ is a gateway from outer rings into ring 1. It should have ring brackets of 1,1,5. However, the Build Script Language offers no mechanism for specifying ring bracket values to be used when installing an Unbound_obj.

Instead, the installer must edit the Build_exec_com command for new_gate_ to add a -ring_brackets or -rb control argument:

```
us add new_gate_                          >tools>== -ss -rb 1 1 5
```

Note that the copy of new_gate_ being installed in the object directory is NOT a gate, and therefore correctly uses the default ring brackets of 1,5,5.

The edited version of the Build_exec_com is shown below.

```
mbuild:  ..pr MCR003.mb.ec

                    MCR003.mb.ec        09/20/19  0817.7 pdt Fri


&version 2
&trace &command off
&-
delete MCR003.mb.i? -brief -query_all
us sd                    -acl re *.*.*  -rb 1 5 5     &- ---- set global defaults ----
&-
&attach
us in MCR003.mb.io -log -no_fill
Test unusual cases:
 - Replacing an unbound segment (a gate).
 - Adding an unbound segment (a new gate).
 - Replacing an archive which is not a component of a bound seg (data_arch).
 - Adding names to an unbound segment (edit build script manually to include the
names).
.
&detach
&-
us in                    -acl re *.*.*                &- -------- executable ---------
us add new_gate_                          >tools>== -ss -rb 1 1 5
us rp  hcs_                               >ldd>hard>execution>== -ss
&-
us in                    -acl r  *.*.*                &- -------- source, object -----
us add new_gate_.alm                      >ldd>tools>source>==
us add new_gate_                          >ldd>tools>object>==
us rp  hcs_                               >ldd>hard>object>==
&-
us ls
us pr
```

Threaded lists used by mbuild for unbound segment installations are shown below. They include an UNBOUNDOBJ structure, which manages installation of Unbound_obj segments directly into library.object and library.execution directories.

```
mbuild:  pr -ls (UNBND source COMP Unb target_only)

UNBOUNDOBJ_Tb:
  UNBOUNDOBJ:              new_gate_           IN: tools  ADD;
                                                   /^  source: new_gate_.alm  ^/
                             add_name:
                               new_priv_gate_
                               new_admin_gate_;
  UNBOUNDOBJ:              hcs_                IN: hard  REPLACE;
                                                   /^  source: hcs_.alm  ^/
source_Tb:
  source:                  hcs_.alm            IN: hard.source  REPLACE compiler: alm;
                                                   /^  COMPILE  UNBOUNDOBJ  ^/
  source:                  new_gate_.alm       IN: tools.source  ADD compiler: alm;
                                                   /^  COMPILE  UNBOUNDOBJ  ^/
COMPILE_Tb:
  COMPILE:                 hcs_.alm            IN: hard.source;
                                                   /^  source: hcs_.alm
                                                       object: hcs_  ^/
  COMPILE:                 new_gate_.alm       IN: tools.source;
                                                   /^  source: new_gate_.alm
                                                       object: new_gate_  ^/
Unbound_obj_Tb:
  Unbound_obj:             hcs_                IN: hard.object  REPLACE;
                                                   /^  COMPILE  ^/
  Unbound_obj:             new_gate_           IN: tools.object  ADD;
                                                   /^  COMPILE  ^/
target_only_Tb:
  Seg(data_arch):          asu_data.archive    IN: tools.execution  REPLACE;
```

Structure: UNBOUNDOBJ

The analyze request may find references to unbound segments in the installation directory. The directory may contain source segments which are found as standalone segments by a library search; or by a Build_script file including an Unbound_obj statement. analyze uses these clues to fabricate an UNBOUNDOBJ structure for each unbound segment being updated.

This UNBOUNDOBJ structure contains:

- a pointer to a Seg(source) structure; and
- pointers to arrays of names to be added to or deleted from the Unbound_obj segment.

```
dcl 1 UNBOUNDOBJ aligned based(UNBOUNDOBJp),
    2 common,           /* Elems in: Seg, COMPILE, BOUNDOBJ, UNBOUNDOBJ, request  */
      3 info aligned like Seg.info,
      3 request_Td aligned like Tlist_data,
                        /*     request.request_Tb --->> Seg.request_Td  (Per Req)  */
    2 Td,
      3 UNBOUNDOBJ_Td aligned like Tlist_data,
                        /*build_data.UNBOUNDOBJ_Tb --->> UNBOUNDOBJ.UNBOUNDOBJ_Tb  */
    2 ptrs,
      3 name_addP ptr,  /* Pointer to list of alternate names on the seg.         */
      3 name_deleteP ptr,
                        /* Pointer to list of names to be deleted from the seg.   */

      3 sourceP ptr,    /*      UNBOUNDOBJ.sourceP  --> Seg(source)               */
    UNBOUNDOBJp ptr;
```

Adding a New Bound Object

A changeset which is creating a new bound segment to add to the library contains:

- A complete source archive containing all source components of the new bound segment.
- A bind file as a separate segment.
- New and/or changed include files used by the source components.
- New and/or changed info segments.

The new bound_mbuild_ is a typical example of such changeset.

```
pwd
>user_dir_dir>Multics>GDixon>w>MCR004
r 16:12 0.069 0

ls -sort

Segments = 15, Lengths = 154.

r w    1  MCR004.mb
r w    1  bound_mbuild_.bind
r w  114  bound_mbuild_.s.archive
rew    7  mbuild.info
          mb.info
          build_script.gi.info
rew    3  mbuild_Tlist_.incl.pl1
rew    1  mbuild_Tlist_dcls_.incl.pl1
rew   16  mbuild_data_.incl.pl1
rew    1  mbuild_display_dcls_.incl.pl1
rew    2  mbuild_info_.incl.pl1
rew    1  mbuild_request_parms_.incl.pl1
r      2  mbuild_type.info
          mbt.info
          mbuild_type.paradigm.info
          mbt.paradigm.info
          mbuild_type.pdm.info
          mbt.pdm.info
          mbuild_type.seg_type.info
          mbt.seg_type.info
          mbuild_type.seg.info
          mbt.seg.info
rew    1  ssu_command_dcls_.incl.pl1
rew    1  ssu_request_dcls_.incl.pl1
rew    2  ssu_standalone_command_.incl.pl1
rew    1  ssu_subroutine_dcls_.incl.pl1

r 16:12 0.165 0
```

The Build_script file starts out containing only a description of the changeset. Build starts with scan, analyze, and print requests.

```
mbuild:  read -desc

Description:
Test adding a new bound object with all its components:
 - bindfile
 - new sources, all present in install dir within their source archive.
 - new Include and Info segments.
    - added names on Info segments.

mbuild:  scan; analyze; print

 Build_script:             MCR004.mb;

 Bound_obj:                bound_mbuild_                    IN: UNKNOWN  ADD;
     bindfile:             bound_mbuild_.bind                ADD;
   source_arch:            bound_mbuild_.s.archive           ADD;
     source:               mbuild.pl1                        ADD   compiler: pl1 -ot;
     source:               mbuild_.pl1                       ADD   compiler: pl1 -ot;
     source:               mbuild_Tlist_.pl1                 ADD   compiler: pl1 -ot;
     source:               mbuild_analyze_.pl1               ADD   compiler: pl1 -ot;
     source:               mbuild_archive_.pl1               ADD   compiler: pl1 -ot;
     source:               mbuild_clean_.pl1                 ADD   compiler: pl1 -ot;
     source:               mbuild_compile_.pl1               ADD   compiler: pl1 -ot;
     source:               mbuild_data_.pl1                  ADD   compiler: pl1 -ot;
     source:               mbuild_display_.pl1               ADD   compiler: pl1 -ot;
     source:               mbuild_et_.alm                    ADD   compiler: alm;
     source:               mbuild_help_.pl1                  ADD   compiler: pl1 -ot;
     source:               mbuild_info_.cds                  ADD   compiler: cds;
     source:               mbuild_info_find_.pl1             ADD   compiler: pl1 -ot;
     source:               mbuild_install_.pl1               ADD   compiler: pl1 -ot;
     source:               mbuild_lib_names_.pl1             ADD   compiler: pl1 -ot;
     source:               mbuild_library_.pl1               ADD   compiler: pl1 -ot;
     source:               mbuild_lpn.pl1                    ADD   compiler: pl1 -ot;
     source:               mbuild_print_.pl1                 ADD   compiler: pl1 -ot;
     source:               mbuild_progress_.pl1              ADD   compiler: pl1 -ot;
     source:               mbuild_request_parms_.pl1         ADD   compiler: pl1 -ot;
     source:               mbuild_request_tables_.alm        ADD   compiler: alm;
     source:               mbuild_scan_.pl1                  ADD   compiler: pl1 -ot;
     source:               mbuild_script_.pl1                ADD   compiler: pl1 -ot;
     source:               mbuild_script_parse_.rd           ADD   compiler: rdc -ot -trace off;
     source:               mbuild_set_.pl1                   ADD   compiler: pl1 -ot;
     source:               mbuild_type.pl1                   ADD   compiler: pl1 -ot;

 Include:                  mbuild_Tlist_.incl.pl1            IN: sss.include  ADD;
 Include:                  mbuild_Tlist_dcls_.incl.pl1       IN: sss.include  ADD;
 Include:                  mbuild_data_.incl.pl1             IN: sss.include  ADD;
 Include:                  mbuild_display_dcls_.incl.pl1     IN: sss.include  ADD;
 Include:                  mbuild_info_.incl.pl1             IN: sss.include  ADD;
 Include:                  mbuild_request_parms_.incl.pl1    IN: sss.include  ADD;
 Include:                  ssu_command_dcls_.incl.pl1        IN: sss.include  ADD;
 Include:                  ssu_request_dcls_.incl.pl1        IN: sss.include  ADD;
 Include:                  ssu_standalone_command_.incl.pl1  IN: sss.include  ADD;
 Include:                  ssu_subroutine_dcls_.incl.pl1     IN: sss.include  ADD;

 Info:                     mbuild.info                       IN: UNKNOWN.info  ADD;
                             add_name:
                               mb.info
                               build_script.gi.info;
 Info:                     mbuild_type.info                  IN: UNKNOWN.info  ADD;
                             add_name:
                               mbt.info
                               mbuild_type.paradigm.info
                               mbt.paradigm.info
                               mbuild_type.pdm.info
                               mbt.pdm.info
                               mbuild_type.seg_type.info
                               mbt.seg_type.info
                               mbuild_type.seg.info
                               mbt.seg.info;
```

When adding a bound segment, all source files contained in the bound source archive are treated as being added as well. scan and analyze prepare to compile these source files, and archive the object segments resulting from those compilations.

Notice that include files being ADDed to the library are assigned to the only library.directory which stores include files. However, other segments being ADDed (two info segments, and the new bound segment) have UNKNOWN library designations. The installer must specify a real library name for those new segments. The installer uses the set command to change these library names.

```
mbuild:  set -seg mbuild("" _type).info -library priv
 Info:                mbuild.info              IN: priv.info  ADD;
 Info:                mbuild_type.info         IN: priv.info  ADD;

mbuild:  set -seg bound_mbuild_ -library tools
 Bound_obj:           bound_mbuild_            IN: tools  ADD;
    bindfile:         bound_mbuild_.bind       IN: tools.object  ADD;
   source_arch:       bound_mbuild_.s.archive  IN: tools.source  ADD;
     source:          mbuild.pl1               IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild.pl1               IN: tools.source;
     source:          mbuild_.pl1              IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_.pl1              IN: tools.source;
     source:          mbuild_Tlist_.pl1        IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_Tlist_.pl1        IN: tools.source;
     source:          mbuild_analyze_.pl1      IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_analyze_.pl1      IN: tools.source;
     source:          mbuild_archive_.pl1      IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_archive_.pl1      IN: tools.source;
     source:          mbuild_clean_.pl1        IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_clean_.pl1        IN: tools.source;
     source:          mbuild_compile_.pl1      IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_compile_.pl1      IN: tools.source;
     source:          mbuild_data_.pl1         IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_data_.pl1         IN: tools.source;
     source:          mbuild_display_.pl1      IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_display_.pl1      IN: tools.source;
     source:          mbuild_et_.alm           IN: tools.source  ADD compiler: alm;
     COMPILE:         mbuild_et_.alm           IN: tools.source;
     source:          mbuild_help_.pl1         IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_help_.pl1         IN: tools.source;
     source:          mbuild_info_.cds         IN: tools.source  ADD compiler: cds;
     COMPILE:         mbuild_info_.cds         IN: tools.source;
     source:          mbuild_info_find_.pl1    IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_info_find_.pl1    IN: tools.source;
     source:          mbuild_install_.pl1      IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_install_.pl1      IN: tools.source;
     source:          mbuild_lib_names_.pl1    IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_lib_names_.pl1    IN: tools.source;
     source:          mbuild_library_.pl1      IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_library_.pl1      IN: tools.source;
     source:          mbuild_lpn.pl1           IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_lpn.pl1           IN: tools.source;
     source:          mbuild_print_.pl1        IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_print_.pl1        IN: tools.source;
     source:          mbuild_progress_.pl1     IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_progress_.pl1     IN: tools.source;
     source:          mbuild_request_parms_.pl1  IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_request_parms_.pl1  IN: tools.source;
     source:          mbuild_request_tables_.alm IN: tools.source  ADD compiler: alm;
     COMPILE:         mbuild_request_tables_.alm IN: tools.source;
     source:          mbuild_scan_.pl1         IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_scan_.pl1         IN: tools.source;
     source:          mbuild_script_.pl1       IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_script_.pl1       IN: tools.source;
     source:          mbuild_script_parse_.rd  IN: tools.source  ADD compiler: rdc -ot -trace off;
     COMPILE:         mbuild_script_parse_.rd  IN: tools.source;
     source:          mbuild_set_.pl1          IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_set_.pl1          IN: tools.source;
     source:          mbuild_type.pl1          IN: tools.source  ADD compiler: pl1 -ot;
     COMPILE:         mbuild_type.pl1          IN: tools.source;
```

The installer saves changes in the Build_script file.

```
mbuild:  save; progress

   COMPLETED   REQUESTS for Builds
   _________   ____________________________
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
               compile
               archive_prep
               install_ec
```

Then build and install tasks continue as in the earlier example. For brevity, these steps are not shown in this bulletin, but may be summarized as follows:

- A compile request compiles all source files.
- An archive_prep request: fabricates the object archive, adding to it the bindfile and the object segments output by the compilations; binds the archive into the bound_mbuild_ segment.
- An install_ec request then creates a Build_exec_com to install these new components into the library.

Updating non-Object Segments

Some changesets include segments not seen in earlier examples. A good example are administrative exec_com files that follow the object_x_only build paradigm: they are installed only in object and execution directories; they involve no source files requiring compilation; no bound segments needing archive updates or binding.

mbuild includes special code to install segments which follow the object_x_only paradigm. These include the segment types shown below:

```
mbuild_type seg -fpdm object_x_only -name -desc

----------
source_starname:              **.ec
description:                  Executable Command File

----------
source_starname:              **.dcl
description:                  PL/I Declaration Definitions

----------
source_starname:              **.ttf
description:                  Terminal Type Definitions File
```

For such changesets, the build/install steps involve only scan, analysis, and install_ec requests. These are shown below.

```
ls

Segments = 3, Lengths = 21.

r w    1  MCR011.mb
r w   10  acct_start_up.ec
rew   10  acct_start_up_1.ec

r 20:15 0.065 0

mb

Installation_directory: >user_dir_dir>Multics>GDixon>w>MCR011

mbuild:  read -desc

Description:
Test installation of exec_coms (object_x_only paradigm):
 - acct_start_up.ec           REPLACE
 - acct_start_up_1.ec         ADD

mbuild:  scan
```

```
mbuild:  print

Segments found by scan request:

  Build_script:            MCR011.mb;

  Seg(exec_com):           acct_start_up.ec            IN: tools.execution  REPLACE;
  Seg(exec_com):           acct_start_up_1.ec          IN: UNKNOWN.execution  ADD;
```

The existing Build_script file was edited earlier to resolve the UNKNOWN library name for the new file (acct_start_up_1.ec) being added. Read in that file to inform mbuild of this change.

```
mbuild:  read

Description:
Test installation of exec_coms (object_x_only paradigm):
 - acct_start_up.ec           REPLACE
 - acct_start_up_1.ec         ADD

Segments found by read request:

  Build_script:            MCR011.mb;

  Seg(exec_com):           acct_start_up.ec            IN: tools.execution  REPLACE;
  Seg(exec_com):           acct_start_up_1.ec          IN: tools.execution  ADD;

mbuild:  analyze; print

 Build_script:            MCR011.mb;

 Seg(exec_com):           acct_start_up.ec            IN: tools.execution  REPLACE;
 Seg(exec_com):           acct_start_up_1.ec          IN: tools.execution  ADD;

mbuild:  progress

   COMPLETED   REQUESTS for Builds
   _________   ____________________________
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
      yes      compile
      yes      archive_prep
               install_ec

mbuild:  install_ec
```

```
mbuild:  ..pr MCR011.mb.ec

                    MCR011.mb.ec        09/20/19  0823.2 pdt Fri


&version 2
&trace &command off
&-
delete MCR011.mb.i? -brief -query_all
us sd                    -acl re *.*.*  -rb 1 5 5     &- ---- set global defaults ----
&-
&attach
us in MCR011.mb.io -log -no_fill
Test installation of exec_coms (object_x_only paradigm):
 - acct_start_up.ec           REPLACE
 - acct_start_up_1.ec         ADD
.
&detach
&-
us in                    -acl re *.*.*                &- -------- executable ---------
us rp  acct_start_up.ec                 >tools>==
us add acct_start_up_1.ec               >tools>==
&-
us in                    -acl r  *.*.*                &- -------- source, object -----
us rp  acct_start_up.ec                 >ldd>tools>object>==
us add acct_start_up_1.ec               >ldd>tools>object>==
&-
us ls
us pr
```

Deleting Segments

The final example involves deleting existing segments from the library. In most cases, the installation directory contains only a Build_script calling for the deletion. There are no segments being added or replaced.

However, deleting a source component from a bound segment usually requires updating that bound segment's bindfile, to remove the object compiled from that source from the Order statement, and perhaps to remove an objectname: group describing the removed object segment.

The changeset in this example shows deleting five kinds of segments from the library.

```
mb

Installation_directory: >user_dir_dir>Multics>GDixon>w>MCR009

mbuild:  read

Description:
Test deletion of:
 - Source Component of Bound_obj
 - Bound_obj
 - Unbound_obj
 - Include
 - Info

Segments found by read request:

  Bound_obj:               bound_hunt_dec_             IN: tools.execution  DELETE;

  bound_pnotice_.archive:
    bindfile:              bound_pnotice_.bind         IN: tools.object  REPLACE;

  bound_pnotice_.s.archive:
    source:                pnotice_mlr_.alm            IN: tools.source  DELETE;

  Unbound_obj:             volume_dump_switch_on       IN: sss.object  DELETE;

  Seg(Include):            cds_args.incl.pl1           IN: sss.include  DELETE;

  Seg(Info):               volume_dump_trace_on.info   IN: priv.info  DELETE;

mbuild:  analyze

mbuild:  print

 Bound_obj:                bound_hunt_dec_             IN: tools  DELETE;

 Bound_obj:                bound_pnotice_              IN: tools  UPDATE;
     bindfile:             bound_pnotice_.bind          REPLACE;
     source:               pnotice_mlr_.alm             DELETE        compiler: alm;

 Unbound_obj:              volume_dump_switch_on       IN: sss  DELETE;

 Include:                  cds_args.incl.pl1           IN: sss.include  DELETE;

 Info:                     volume_dump_trace_on.info   IN: priv.info  DELETE;
```

```
mbuild:  progress

   COMPLETED   REQUESTS for Builds
   ---------   ---------------------------
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
      yes      compile
               archive_prep
               install_ec


mbuild:  archive_prep

---------- library_fetch bound_pnotice_.s.archive -lb tools.source

   archive d bound_pnotice_.s.archive pnotice_mlr_.alm


---------- library_fetch bound_pnotice_.archive -lb tools.object

   archive u bound_pnotice_.archive bound_pnotice_.bind

   archive d bound_pnotice_.archive pnotice_mlr_


----------  bind  bound_pnotice_.archive

Binding bound_pnotice_


mbuild:  pg

   COMPLETED   REQUESTS for Builds
   ---------   ---------------------------
      yes      set -desc   or    read -desc
      yes      scan      and/or  read
      yes      analyze
      yes      compile
      yes      archive_prep
               install_ec


mbuild:  install_ec
```

```
mbuild:  ..pr MCR009.mb.ec

                    MCR009.mb.ec        09/20/19  0838.4 pdt Fri


&version 2
&trace &command off
&-
delete MCR009.mb.i? -brief -query_all
us sd                     -acl re *.*.*  -rb 1 5 5     &- ---- set global defaults ----
&-
&attach
us in MCR009.mb.io -log -no_fill
Test deletion of:
 - Source Component of Bound_obj
 - Bound_obj
 - Unbound_obj
 - Include
 - Info
.
&detach
&-
us in                     -acl re *.*.*               &- -------- executable ---------
us dl                                    >tools>bound_hunt_dec_ -ss
us rp  bound_pnotice_                    >tools>== -ss
us dl                                    >sss>volume_dump_switch_on -ss
&-
us in                     -acl r  *.*.*               &- -------- source, object -----
us dl                                    >ldd>tools>source>bound_hunt_dec_.s.archive
us dl                                    >ldd>tools>object>bound_hunt_dec_.archive
us rp  bound_pnotice_.s.archive          >ldd>tools>source>== -ac
us rp  bound_pnotice_.archive            >ldd>tools>object>== -ac
us dl                                    >ldd>sss>source>volume_dump_switch_on.pl1
us dl                                    >ldd>sss>object>volume_dump_switch_on
&-
us in                     -acl r  *.*.*               &- -------- Include ------------
us dl                                    >ldd>include>cds_args.incl.pl1
&-
us in                     -acl r  *.*.*               &- -------- Info ---------------
us dl                                    >doc>privileged>volume_dump_trace_on.info
&-
us ls
us pr
```

Using mbuild with a Private Library

The mbuild design described in this bulletin focuses on program builds and installations for the Multics Libraries, which present Multics software programs to users. Can private libraries also make use of mbuild to build and install their software? The answer is YES, if the private library meets certain requirements.

Library Requirements

- Segment Types: the library must contain segment types that mbuild knows how to build and install. For a list of known segment types, enter the command: mbuild_type segs

- Library Structure: the library must share the directory structure used by the Multics Libraries, with each library including directories named:
  source, object, execution, include, info, listings

- Build/Install Paradigms: the segment types must follow the Multics Libraries model for building and installing segments into library directories: policies for naming segments, organizing components of bound segments, etc. For a brief summary of those paradigms, type:
  mbuild_type pdm -all

Creating a Private Library Descriptor

mbuild obtains information about library structure from a database called a library descriptor. For detailed information about library descriptors, see the manual:
Multics Library Maintenance Program Logic Manual (AN80)

The following sample library descriptor defines a single library containing the six required directories named above:

```
pr gcd_libraries_.ld

                    gcd_libraries_.ld   08/03/19  0940.5 pdt Sat


Descriptor:                  gcd_libraries_;


Define:                      commands;
     command:                library_info;
          library name:      gcd_library;
     command:                library_map;
          library names:     gcd_library.source gcd_library.object gcd_library.execution;
          search name:       **;
     command:                library_print;
          library name:      info;
          search name:       **.*.info;
     command:                library_fetch;
          library names:     gcd_library.source gcd_library.include gcd_library.info;
     command:                library_cleanup;
          library name:      gcd_library;
          search name:       !?????????????;
```

```
Root:                       (gcd_library gcd "").(source s "");
     path:                  >udd>m>gd>lib>source;
     search procedure:      multics_library_search_$source_dirs;

Root:                       (gcd_library gcd "").(object o "");
     path:                  >udd>m>gd>lib>object;
     search procedure:      multics_library_search_$object_dirs;

Root:                       (gcd_library gcd "").(listings);
     path:                  >udd>m>gd>lib>listings;
     search procedure:      multics_library_search_$list_info_dirs;

Root:                       (gcd_library gcd "").(execution x "");
     path:                  >udd>m>gd>bin;
     search procedure:      multics_library_search_$execution_dirs;

Root:                       (gcd_library gcd "" lang).(include incl "");
     path:                  >udd>m>gd>lib>include;
     search procedure:      multics_library_search_$list_info_dirs;

Root:                       (gcd_library gcd "").(info "");
     path:                  >udd>m>gd>info;
     search procedure:      multics_library_search_$list_info_dirs;

End:  gcd_libraries_;
```

Create a descriptor for your library patterned after the above. The path statements may specify any directory structure and location; the library directories need not be subdirectories of a single root, as the above example demonstrates. Create those directories, ensuring the mbuild user has sma access to install segments into each directory.

Use the library_descriptor_compiler (ldc) command to compile your descriptor. Install its output object segment in one of the directories named in your linker search paths.

Using the Private Library Descriptor

The library_descriptor (lds) command sets the default library descriptor used in a Multics process:

```
lds set gcd_libraries_
```

The library tools seem to work best if the same descriptor is used throughout the life of the process. If you want to use another descriptor, select it in a new process rather than issuing a second lds set command in the current process.

mbuild uses the library tools to obtain information from the default library descriptor. If you set a private descriptor before invoking mbuild, it will then get data for this descriptor. If unsure about which descriptor is in use, type: lds name

Use the mbuild libs request to see mbuild's preferred names for the library directories in your descriptor. These preferred library names should be use when adding segments to your library.

Using update_seg for a Private Library

mbuild also uses the update_seg command to install segments into the target library. Normally, update_seg uses the >tools>installation_tools_ gate to install segments into an execution ring lower that the installer's login ring. However, if the installer does not have access to the installation_tools_ gate, then update_seg uses the >hardcore>hcs_ gate to install segments into the private library at the login ring level.

NOTE: When ADDing segments to a private library, it may be necessary to add a -ring_brackets option to the update_seg set_defaults commands in the Build_exec_com. For example:

```
us sd    -acl re *.*.*  -rb 4 5 5       &- ---- set global defaults ----
```

Possible Enhancements to mbuild

mbuild could be enhanced to provide services for code developers and auditors by interfacing with more of the tools listed in Table 2. For example:

- display_pnotice: mbuild knows which segments are part of the changeset, and which of these are source programs that should contain protection notices. It could provide a request to run display_pnotice on each source segment.

- validate_info_seg, vis: mbuild knows which segments in the install directory are info segments. It could provide a request to run validate_info_seg against all new/changed info segs.

- include files changes: mbuild could be trained to use the crossref database to identify which segments need to be rebuilt to accommodate a change to an include file.
    - Such feature might start out as a crude enumeration of all segments using the include file, giving the ability to recompile those segments (without modifications) to see if include file changes introduce any compiler-detected errors in the including source segments.

Time constraints prevented full tailoring of the ssu_ environment to support mbuild. The following changes might be useful:

- list_help: add an mbuild version of list_help, tailored to report available help files from the three locations searched by the mbuild help command.

Appendix A: Build Script Language in BNF

```
Backus-Naur Form for the Build Script Language:

    · { <alt-1> | <alt-2> | <alt-3> } identifies alternatives.  Choose only one of them.
      { <alt-1> | <alt-2> | <alt-3> }... says choose one or more of the alternatives.

    · [....] says stuff is optional.  [....]... says choose zero or more of this item.

   ----------                                                                  ----------

<script-contents>       ::= <description-group> [<mb-script-item>]... <installable-item>...

<description-group>     ::= <description-stmt> <install-dir-stmt> | <install-dir-stmt>

 <description-stmt>     ::= Description: <description-line>...

 <install-dir-stmt>     ::= Installation_directory: <absolute-pathname> ;


<installable-item>      ::= {<bound-object-group>|<unbound-object-group>|
                             <info-seg-group>|<include-file-group>|<unanalyzed-seg-group>}

<bound-object-group>    ::= <bound-object-stmt> [<bindfile-stmt>] [<archive-source-group>]...

 <archive-source-group> ::  {<source-stmt>|<archive-stmt> <source-stmt>...}

<unbound-object-group>  ::= <unbound-object-stmt> <source-stmt> [<naming-group>]...

<include-file-group>    ::= <include-file-stmt> ;

<info-seg-group>        ::= <info-seg-stmt> [<naming-group>]...

<unanalyzed-seg-group>  ::= <unanalyzed-seg-stmt> [<naming-group>]... ;


<bound-object-stmt>     ::= Bound_obj: <bound-object-name> <library> <bound-obj-operation> ;

  <library>             ::=  IN: <library-name>

  <bound-obj-operation> ::=  {ADD|UPDATE|DELETE}

<bindfile-stmt>         ::=  bindfile: <bind-file-name> [<library>] <operation> ;

  <operation>           ::=  {ADD|REPLACE|DELETE}

<archive-stmt>          ::=  source_arch:  <bound-source-archive-name> [<library>]
                                              <bound-obj-operation> ;

<source-stmt>           ::=  source:   <source-seg-name> [<library>] <operation>
                                         [<compiler-group>]

  <compiler-group>      ::=   <compiler> [<compile-option>]... ;

  <compiler>            ::=   compiler: <compiler-name>

  <compile-option>      ::=   {<control-arg>|<control-arg> <value-word>}


<unbound-object-stmt>   ::= Unbound_obj: <object-seg-primary-name> <library> <operation> ;

<include-file-stmt>     ::= Include: <include-file-name> [<library>] <operation>;

<info-seg-stmt>         ::= Info: <info-seg-primary-name> <library> <operation> ;

<unanalyzed-seg-stmt>   ::= Seg(<seg-type>): <seg-name> <operation> ;
```

```
<naming-group>          ::= <add_name-stmt>|<delete_name-stmt>
 <add_name-stmt>        ::=  add_name: <add-seg-name>... ;
 <delete_name-stmt>     ::=  delete_name: <del-seg-name>... ;

<mb-script-item>        ::= {<mb-script-stmt>|<mb-exec_com_stmt>|<mb-io-stmt>|<mb-log-stmt>}
 <mb-script-stmt>       ::= Build_script: <mb-seg-name>.mb ;
 <mb-exec_com-stmt>     ::= Build_exec_com: <mb-seg-name>.mb.ec ;
 <mb-io-stmt>           ::= Build_io: <mb-seg-name>.mb.io ;
 <mb-log-stmt>          ::= Build_log: <mb-seg-name>.mb.il ;
```