To: Distribution

From: C. D. Tavares

Date: 12/07/77

Subject: New Cross-Referencer

This MTB proposed a replacement for the currently existing object segment (library) cross-referencer. The cross-referencer that exists has several major problems and a few minor flaws. In addition, a few desirable features are proposed for the new package. Sample MPM-style documentation is attached.

## PROBLEMS WITH THE CURRENT CROSS-REFERENCER

The largest, and most visible problem with the current cross-referencer is that it no longer will cross-reference the entire system, as a result of the storage management it performs being limited to a single segment worth of internal data.

The current cross-referencer attempts to perform what is necessarily a two-pass operation (first scanning all the known entries and definitions, and then associating link references with them) in one pass. This results in the current cross-referencer producing a fair number of misinformed correlations.

The current cross-referencer is unable to comprehend various forms of duplicate segment names that it may encounter in its meanderings. No matter how many times distinct instances of the same module (or different modules with the same name) appear among the items it is searching, the cross-referencer will lump them together as if they were one module. Whether it is desirable or not that modules in Multics have totally unique names is a moot point. The fact is that they exist (e.g., tape_checksum_, status, trace) and they must be identified correctly.

The current cross-referencer has no concept of how bindfiles may alter the apparent external availability of entrypoints. Such things as internal or or added names in critical places can have major consequences.

One minor but ubiquitous annoyance is that the cross-referencer sorts its cross-reference in REVERSE alphabet-

Multics Project working documentation. Not to be reproduced or distributed outside the Multics Project.

ical order.

Lastly, the code as it exists is not maintainable. It walks archives and object segments by itself instead of using the proper tools. It manages its storage as a large array of words, when in reality it is a large collection of variable-sized structures. In addition, none of the references are mnemonic to aid the maintainer in figuring out what exactly is represented by any piece of data that is being manipulated.

## DESIRABLE FEATURES

The cross-referencer should keep information about where each module was located. It should use that information to produce best-fit matches for references. The package should first attempt to fulfill the link within the same bound segment (to account for purely internal synonyms and module names); then, that failing, within the same library (defined as a directory or a set of directories); then, that failing, from among the "known universe" of modules.

It is necessary for the cross-referencer to report information such as where a module was located (bound segment and library). For example, it is useful to know that the "create" command is in bound_fscom2_ in SSS. The report should also contain a COMPLETE cross-reference of synonyms and added names. One should not have to "know", for example, that get_pdir_ is really get_process_id_$get_pdir_ when one wants to find it in the cross-reference.

The current cross-referencer allows no way to cross reference an archive or a set of segments without the user producing a driving file. There should be a simpler command interface (i.e. specification of pathnames and starnames) that would allow this type of operation for simple applications. The lack of such a facility presently discourages almost all users from using the cross-referencer for their own private purposes. Its inclusion would make the cross-reference package a generally-useful facility from the viewpoint of sites and users.

The cross-referencer should also be able to produce include file cross-references. All the data necessary to do this is already in the object segment. It would be a much more foolproof method of producing this information than source scanning programs. In addition, it can also identify modules using potentially different versions of the same include file.

It should be possible to specify to the cross-referencer that the output produced should be limited to only those modules that actually participated in the cross-reference. It is very often useful to know the interrelationship of a specific set of modules in a subsystem (e.g., the PL/I compiler) without caring that any of them call hcs_ or com_err_, or so on.

Significant error conditions encountered while performing the cross-reference should be sent to the user's terminal as well as to the output file. For example, the fact that someone referenced a definition in pds$ which is KNOWN not to exist is important enough to warrant immediate notification to the user performing the cross-reference. (Of course, the user should be able to optionally suppress these messages.)

## BENEFITS

A cross-referencer with the described properties has the side effect of validating a large part of the "cleanliness" of the system libraries. It is able to detect missing bindfiles, untrimmed archives (archives without archive names on them, which occur when a new version of an archive with fewer names than the original is loaded without trimming), clobbered archives or object segments, accidental duplications, and a few other problem conditions.

The output of the cross-referencer, as described, gives a very complete representation of the total state of the system libraries. In fact, it has been used by one customer to keep microfiche up to date, including determining which ware obsolete and to be discarded, and which had never been supplied. Operation of the cross-raferencer on the MR4.0 system revealed about twenty discrepancies in the system libraries (mostly as shipped), many of them malformed or damaged segments.

Name: cross_reference, cref

The cross_reference command is a utility for creating a cross-reference listing of any number of object programs. The listing contains information about each object module encountered, including the location of each program, its entrypoints and definitions, its synonyms if any, and which other modules encountered reference each entrypoint or definition. It will also optionally supply a cross-reference listing of include files used by modules encountered.

Usage:

```
cross_reference library_desc1 ... library_descN
                -control_args-
```

library_desc*i* are library descriptions of one of the three forms:

```
pathi path2 ... pathN

-library library_name path1 path2 ... pathN

-library library_name -all path1 path2 ... pathN
```

(The control argument "-library" may be abbreviated as "-lb".) Each path*i* is the pathname of a segment to be examined and cross-referenced. The star convention is honored. The library_name, if present, may be any user-chosen identifier. All modules represented by path1 ... pathN will be treated by the cross-referencer as if they were in a common library directory of that name. If the library description contains the control argument "-all", all the module names encountered will be considered external (see the section on "Resolving References".) This control argument is provided especially for cross-references of the Multics Hardcore libraries.

-control_args- may be one or more consistent combinations of the following control arguments:

| | |
|---|---|
| -input_file filename<br>-if filename | specifies that a control file describing the modules to be cross-referenced is to be used instead of the arguments library_desc_i. If the suffix ".crl" is not part of the supplied filename, it will be assumed. If this control argument is used, no library_desc_i arguments are allowed. |
| -output_file filename<br>-of filename | specifies that the cross-reference list is to be created in a segment of the specified name. If the suffix ".crossref" is not part of the supplied filename, it will be assumed. If this control argument is not supplied, but the -input_file control argument is supplied, the output file will take its name from the input file, with the suffix ".crossref" replacing the suffix ".crl". Otherwise, the output file will be named "crossref.crossref". |
| -brief, -bf | specifies that non-fatal error messages to the terminal are to be suppressed. This control argument does not affect the reporting of error messages to the output file. |
| -first | specifies, when the -input_file control argument has been given, that once any instance of a particular module has been located, the cross-referencer need not search the remaining directories for other instances of modules with the same name. If this control argument is omitted, the cross-referencer will search all libraries in the search list for each module name supplied. |

| | |
|---|---|
| -include_files<br>-icf | specifies that include files used by all modules examined are also to be cross-referenced. |
| -short, -sh | specifies that referenced modules which are not included in the scope of any library_desci should not be included in the output. This control argument causes the output to reflect only the interrelationships among the modules in the libraries specified. |

## Module Examination

Module examination is performed in two passes. The first pass defines all the segment names, synonyms, and definitions. The second pass examines external references, and attempts to resolve them with existing definitions.

Segments encountered fall into four classes: non-object, bound segments, stand-alone modules, and archives.

When a non-object segment is encountered, a warning message is printed to that effect, and the segment is included in the results of the cross_reference.

When a bound segment is encountered, a warning message is printed to that effect, and the segment is ignored. Bound segments are of no use to the cross-referencer, since information necessary to determine which components make use of which external reference links is no longer available due to the binding process. Instead, the object archive from which it was bound should be used.

When a stand-alone segment is encountered, it is analyzed for entrypoints, definitions, and external references. All additional names on the segment are entered as synonyms for the module. This information is then included in the results of the cross-reference.

When an archive is encountered, each component is analyzed for entrypoints, definitions, and external references. If a bindfile exists, synonyms for each component are derived from "synonym:" statements in the bindfile, when they exist. This information is then included in the results of the cross-reference.

Modules are also identified by the segment in which they were found (either themselves, for a stand-alone segment, or the containing archive, for an archive component) and by the library_name of the directory in which they were found. If the directory was specified without a library_name, the pathname of the directory is used as the library_name. This makes it possible to have multiple occurrences of segments with the same name, as long as they differ by at least one of these identification criteria.

## Resolving References

When a module is examined by the cross-referencer, its name and synonyms are classified as "internal" or "external" by the following criteria:

1) If the module is stand-alone, its name and synonyms are external.

2) If the module is archived, and the library description contained the "-all" control argument, its name and all its synonyms are considered external.

3) If the module is archived, and the library description did not contain the "-all" control argument, its name and each of its synonyms is external only if it appears in the "Addname:" statement of the bindfile. If no bindfile exists, the name and synonyms are considered internal.

The cross-referencer attempts to resolve external references on a best-match basis by using the following criteria:

1) If the reference may be satisfied by a definition in the same module, that definition is used.

2) If the referencing module is part of a bound segment, and it may be satisfied by a definition in the same bound segment, that definition is used.

3) If the reference may be satisfied by a external definition in the same library_name, that definition is used.

4) Otherwise, the first external definition encountered which satisfies the reference is used. If more than one such definition exists, a warning message is printed.

## Format of a Driving File

If the -input_file control argument is specified, the cross-referencer takes its input from a special file.

The first lines of the file must contain the names of one or more directories to be searched. They are specified in the following manner:

```
-library:         (OR  -library -all:)
pathname_1              library_name_a
pathname_2              library_name_b
....
pathname_N              library_name_z;
```

Each pathname_i specifies a directory to be searched. Each library_name (which may contain spaces) if present, will be used to describe the preceding directory name. (See "Module Examinatior", above.) The tokens "-wd" or "-working_directory" may be used to represent the current working directory. A semicolon ends the search list.

The next information in the file is a list of the segments to be examined. They must appear one to a line.

If the user wishes to explicitly define synonyms for any modules which would not otherwise be generated (e.g. a non-apparent reference name by which a segment is sometimes initiated), he may include in this section one or more lines of the form:

```
modulename syn1 syn2 ... synN
```

These lines will not by themselves cause the cross-referencer to search for the module "modulename", since it may not be a free-standing segment. Any synonyms defined in this manner are considered external.

A file may consist of several repetitions of the format described above; that is, a search list, segment names, another search list, more segment names, etc. Whenever a new search list is encountered, it replaces the old search list. If a driving file is to be used, the greatest efficiency can be gained by having it consist of multiple occurrences of a one-directory search list followed by the segments contained in that directory.

For example, a control file constructed to cross-reference a student subsystem might look like the following:

```
-library:
>udd>Class>systemdir>object  CLASS SUBSYSTEM;

class_login_responder.archive
class_tests.archive
student_grades_database
audit_procedure
class_utilities.archive
unallowed_compiler_stub fortran pl1
unallowed_compiler_stub
```

## Special Cases

Segments with unique names and segments whose last component is a single digit are ignored, since these are conventions used by the system library tools to denote segments which are to be deleted shortly.

Archives whose names are identical with the exception of a different numeric next-to-last component are considered the same archive.

Definitions or entrypoints in archive components which masquerade as segment names by the expedient of an added name on the bound segment, without benefit of being defined as a synonym for their containing component, will not be cross-referenced satisfactorily.

## Include Files

The cross-reference listing of include files, when requested, is appended to the regular output of the cross-referencer. Each include file encountered is classified by its entryname and its date/time modified. This ensures that modules which use different versions of the same include file will be apparent.

## Example

The following command will produce a cross_reference listing of the Standard Service System in the file "standard.crossref":

```
cref -library STANDARD >ldd>sss>o>** -of standard
```

To produce a cross_reference listing of the hardcore library, the following command may be used:

```
cref -library HARD -all >ldd>h>o>*  >ldd>h>bc>*.archive
-of hard
```

(Note the use of the "-all" control argument.)

## Output Example

Entries in the output listing are separated by dashed lines. The following is a sample entry:

```
---------------     ***** bound_x_ in SSS *****    ---------------
sample_segname       SYNONYM: one_syn, another_syn
 one_entrypoint      program_a program_b
 second_entrypoint   program_a program_c
 unused_entrypoint
 undefined_ent (?)   program_d
```

The entry shown is for segment "sample_segname", which is a component of bound_x_ in the library specified as SSS. It possesses three entrypoints: "one_entrypoint", "second_entrypoint", and "unused_entrypoint". The information shows that "sample_segname$one_entrypoint" is called by module "program_a" and module "program_b". The question mark after entrypoint "undefined_ent" signifies that this entrypoint is an implicit definition; that is, that module "program_d" references "sample_segname$undefined_ent", but that entpoint does not actually exist. (A diagnostic is printed when this situation is encountered.)

All error messages produced during the run, including warning messages which may not have been printed at the terminal due to use of the "-brief" control argument, are appended to the end of the output file for reference.