NASA/TM—2011-216948

Symbol Tables and Branch Tables

Linking Applications Together

Louis M. Handler
Glenn Research Center, Cleveland, Ohio

January 2011

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to *help@sti.nasa.gov*

- Fax your question to the NASA STI Help Desk at 443–757–5803

- Telephone the NASA STI Help Desk at 443–757–5802

- Write to:
  NASA Center for AeroSpace Information (CASI)
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/TM—2011-216948

Symbol Tables and Branch Tables
Linking Applications Together

Louis M. Handler
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

January 2011

Trade names and trademarks are used in this report for identification only. Their usage does not constitute an official endorsement, either expressed or implied, by the National Aeronautics and Space Administration.

Level of Review: This material has been technically reviewed by technical management.

Available from

NASA Center for Aerospace Information
7115 Standard Drive
Hanover, MD 21076–1320

National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Available electronically at http://gltrs.grc.nasa.gov

Contents

List of Figures

Symbol Tables and Branch Tables
Linking Applications Together

Louis M. Handler
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

1.0 Introduction

While working on the Space Telecommunications Radio System (STRS) reference implementation, the author used techniques that he had not used since before PCs with virtual memory were invented. The application for STRS was to allow parts of the program to be loaded later as specified by the user. This document was written to describe some old-fashioned techniques for connecting software components including branch tables and indirect address tables.

This document explores the computer techniques used to execute software whose parts are compiled and linked separately. In such cases, there may be problems in using the external methods and data defined in each part from the other part. Linking the parts together may not be the best solution. This paper examines alternatives which may be better under certain circumstances. The computer techniques include creating a branch table or indirect address table and using those to connect the parts. Methods of storing the information in data structures are discussed as well as differences between C and C++.

2.0 Symbol Table and Compilation

To execute a program beginning with source files, the user must do compiling, linking, and loading. Compilers often do both compiling and linking, which also may be called building. Compiling is the process of transforming source files into relocatable object files. Linking is the process of assembling object files, including those in libraries, usually into an executable file. Loading is the process of copying the executable file to memory and starting execution.

When a C language source file is compiled into an object file, a function name or data name in C is transformed into a relocatable symbol often using the same name, possibly with an underscore added, as well as storing information about the function or data's relative location and size. A method name in C++ is transformed differently because one can use the same name in different classes and/or with different calling sequences (signatures) for method overloading. For example, the method setBT(AClass* , struct ABranchTable*) in class BClass would have a relocatable symbol name containing information for all of the above, possibly something like: _6BClass_5setBT_p6AClass_p12ABranchTable_. For one DIAB compiler, the symbol name was _setBT__6BClassSFP12ABranchTable. This is known as name mangling. These symbol names are different for different compilers. The linker uses the symbol names to combine object files and adjust relative locations. The loader uses the symbol names to specify where to start execution, for error messages, and debugging.

The programmer normally can rely on the compiler for association of the source code names with their locations. However, when the programmer loads a separate part of the executable during execution, he/she may need to determine the exact name transformation and use it to associate the corresponding symbol name as loaded with its location in memory. In some systems, there are functions to aid the programmer in manipulating the symbol tables that use either the original name or the mangled name as its argument. Then the user can associate the original name with its location. In VxWorks, functions such as LoadModuleAt and symFindByName or symEach may be used. In Linux, functions such as dlopen, dlsym, and dlclose may be used. On systems using the GNU compilers, the GNU libtools may be used. In Windows MSDN, SymLoadModuleEx, GetProcAddress, and related functions may be used. If none of these techniques is available, but the file is an ELF file, the user may need to parse the ELF file (see Section 6.0), perform the relocation, and create a symbol table.

3.0 Branch Table

A branch table (or jump table) is a sequence of unconditional branch (go to) commands used to redirect to the appropriate address a call to a method. Note that branch tables are implemented using an Assembly Language technique for calling C language functions (or their equivalent) indirectly. From the perspective of a caller loaded separately from the callee, this technique has the effect that the first line of each method is executed in the branch table and subsequent lines are executed in the original function location. From the perspective of the callee, only the lines in the original function location are executed. Thus, the method can be executed equivalently by calling (branch and link to) either the original function location or the location in the branch table. The branch table must be formed by the callee and shared with the caller. In the code structure for each object (caller versus callee) the function names are used differently. In the sequence of branches within the caller, the function names must be used as the names of the labels. Since the branch locations are defined within the callee, the caller must merely reserve space. For example, pseudo-code to reserve space:

```
struct caller_reserves_space_for_branch_table
{
        …
        Methodx_Start  data
        …
}
```

where “data” merely represents the need to reserve space. The sequence of branches within the callee must unconditionally branch to the named functions without changing the registers. For example, pseudo-code to define the branch commands:

```
struct callee_defines_branch_table
{
                …
                goto     Methodx_Start
                …
}
```

Note that here the function names are used as operands of the “goto” instructions. Note also that (once the sharing of the branch table is complete) the underlying assembly code is identical in the caller and the callee. Indeed, these sequences must be referenced at the same location (as described in Section 5.0 below).

When the caller calls Methodx_Start, the name defined in its table is found, the goto at that location is executed, which branches to the Methodx_Start in the callee. The registers are unchanged so that the arguments are passed properly, any return value is unchanged, and the next executable command in the caller will be processed after Methodx_Start has returned.

4.0 Indirect Address Table

An indirect address table (or dispatch table or virtual method table) is a sequence of memory locations used to redirect a call to the appropriate address. This is the method used in the STRS reference implementation. In C++, a sequence of addresses (i.e., pointers to functions) to be called is put into a structure and passed as an argument to the object that needs those addresses. Using the addresses in the structure requires that the call be modified to use the structure. If a pointer to the branch table structure is called brtb and the method name to call is Methodx_Start, then instead of calling Methodx_Start(), one would call brtb->Methodx_Start(). If all the calling sequences are the same, an alternative is to use an array, for example, calling brtb[iStart]().

One might define SPT12 as the type for the Methodx_Start method with a return type of “result_type” and two arguments:

```
typedef result_type    (*SPT12)   (arg1_type , arg2_type );
```

The structure in the object that knows the names and locations of the methods might be:

```
struct ABranchTable {
```

containing, for example, the variable to store the pointer to Methodx_Start in the structure:

```
      SPT12    Methodx_Start;
      …
};
```

An instance of the branch table structure is specified:

```
struct ABranchTable brtb;
```

and the pointer to Methodx_Start stored into the structure:

```
brtb->Methodx_Start = &Methodx_Start;
```

Then the structure containing all the pointers to the methods, brtb, is passed as an argument to the object that needs those methods. Then the object receiving brtb can execute the referenced methods by calling:

```
brtb->Methodx_Start(arg1,arg2);
```

5.0 Resolving Symbol Location

If the loading process resolves the symbols of two objects or applications for calls in both directions, none of this analysis applies and the methods can be called normally as shown in Figure 1.

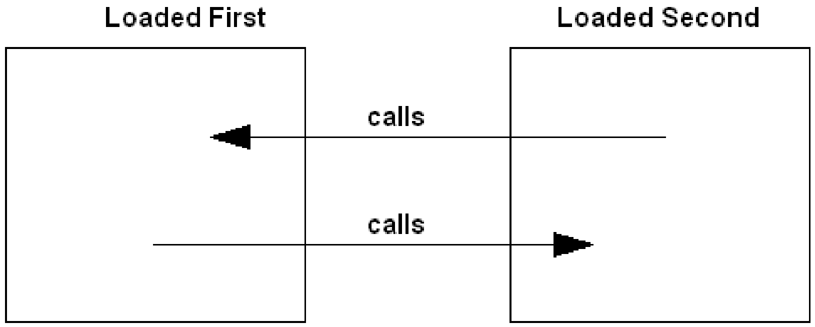

Figure 1.—Symbols resolved in both directions.

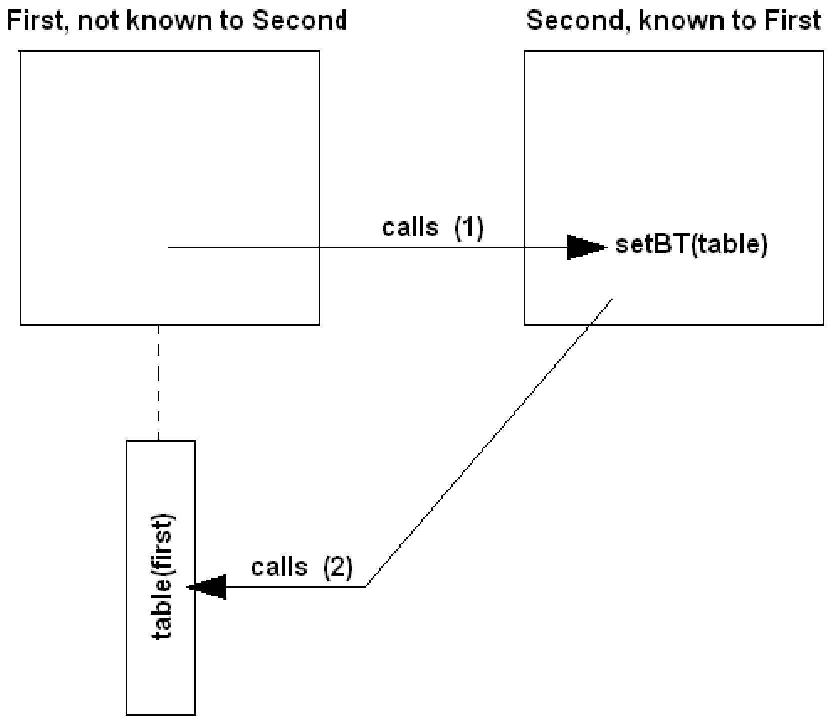

Figure 2.—Symbols resolved in one direction.

If the loading process resolves the symbols in one direction (from “First” to “Second”), a branch table, indirect address table, table of ***this*** pointers, or the equivalent can be passed from “First” to “Second” by adding (and calling) a method to “Second” to receive and store the table. This table contains information for calling the methods in the other direction (from “Second” to “First”). In the example shown in Figure 2, the method in “Second” used to receive the table is named setBT and subsequent calls from the “Second” object/application to the “First” object/application use the table(first) stored by setBT to call the methods indirectly.

If symbols are not resolved in either direction, at least one branch table, indirect address table, table of ***this*** pointers, or the equivalent must be loaded at a predefined location so that the table can be found and the appropriate addresses used. That table will resolve addresses in only one direction. To resolve the addresses in the other direction, either of the previously described methods may be used; i.e., adding a method to receive the other table or storing it in another predefined location (see Figure 3). Usually, the first predefined location is at the beginning of the first compiled, linked, and loaded part so it is always at the beginning of user memory.

In the STRS reference implementation, the loader resolved the symbols in only one direction such that the objects/applications loaded first could use the symbols for those objects/applications that are loaded later but not the reverse, i.e., the objects/applications loaded later could not use the symbols for the previously loaded objects/applications.

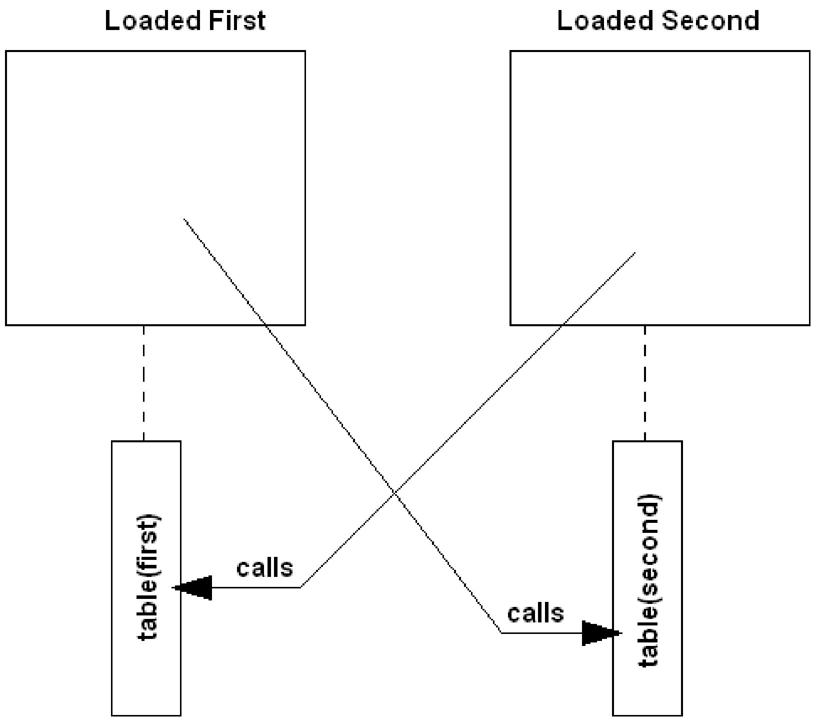

Figure 3.—Symbols not resolved in either direction.

For C language functions or the equivalent C++ methods (using extern "C") defined in the objects/applications loaded first, the table of functions loaded first is passed to the objects loaded second as a structure. As an example, let the method to pass the branch table be setBT in the class whose instance pointer is ***later***. The actual call (from the object loaded first) contains the table of methods, brtb, for example: later->setBT(brtb). Then setBT receives the table of methods and stores it as brtb in the object loaded second so that it can call methods in the object loaded first, using the branch table, for example: (brtb->Methodx_Start)(arg1,arg2). Using Methodx_Start in the branch table as described above only works when the target methods do not require a ***this*** pointer.

However, calling a C++ language instance method does require using its ***this*** pointer. Note that for calls within the same class, the ***this*** pointer is used automatically. As an example of how to handle a C++ application, a method to pass a single ***this*** pointer, setThis, can be added to the class whose instance pointer is ***later***. The object loaded first calls as: later->setThis(this). Then setThis receives the ***this*** pointer of the caller and stores it as ***that*** of the appropriate object type so that the object loaded later can call the object loaded first, using the ***this*** pointer of the object loaded first, for example:

that->Methodx_Start(arg1,arg2).

One problem arose when saving multiple ***this*** pointers to be used to call the same method in objects of different types. They could not be saved as a single object type unless these were in the same hierarchy (object-oriented inheritance relationship). One solution was to save multiple tables of ***this*** pointers, one for each possible object type. Another solution was to add stubs (i.e., methods that do nothing but satisfy the interface) until there is only a single object type for the ***this*** pointers. The stubs would be defined as virtual for all object types but not as pure virtual. (In object-oriented programming, a virtual method is a method whose behavior can be overridden within an inheriting class by a function with the same signature. A pure virtual method is a virtual method that is required to be implemented by a derived class.) The latter was chosen for the STRS reference implementation as simpler with the consideration that it might help eliminate the need for the programmer to implement all the required methods. Also it can simplify the hierarchy as specified in the Motor Industry Software Reliability Association (MISRA) standards, which are often referenced by air and space product companies.

6.0 ELF Format Processing

In the unlikely case that commands are not available to load the executable and resolve symbol tables, the user may have to write them. Many executables are stored in ELF format including those produced by gcc. Here is some basic information about ELF files. The following was condensed from the Linux Journal at URL: http://www.linuxjournal.com/article/1060 .

Note that you may be able to use readelf and objdump to read parts of an ELF file to verify the structure. The names shown below were specific to a particular Linux implementation. For other implementations, check the appropriate elf.h header file.

Each ELF file starts with an ELF header (e.g., ***struct elfhdr*** in /usr/include/linux/elf.h). The ELF header begins with the magic number in hex: 7f 454c46 where the second, third, and fourth bytes are "ELF" in ASCII. The ELF magic number is used to identify this as an ELF file. Following this are fields identifying the type, machine, version, and size of the header, etc.

Each ELF header contains a table that describes the sections within the file. This table contains the ***shnum*** field that indicates how many sections and the ***shoff*** field that indicates the byte offset at which the section header table starts. The ***shentsize*** field indicates the size in bytes of the entry for each section. Each section header is a ***struct ELF32_Shdr***. The name field within this struct is just a number—this is not a pointer, but an offset into the ***.shstrtab*** section (find the index of the ***.shstrtab*** section from the file header in the ***shstrndx*** field). Thus we find the name of each section at the specified offset within the ***.shstrtab*** section.

When you run an ELF program, the kernel looks through the binary and loads it into the user's virtual memory. If the application is linked to a shared library, the application will also contain the name of the dynamic linker that should be used. The kernel then transfers control to the dynamic linker, not to the application. The dynamic loader is responsible for first initializing itself, loading the shared libraries into memory, resolving all remaining relocations, and then transferring control to the application.

The ***.interp*** section simply contains an ASCII string that is the name of the dynamic loader. In Linux, this will be /lib/elf/ld-linux.so.1 (the dynamic loader itself is also an ELF shared library).

The ***.hash*** section is just a hash table that is used so that we can quickly locate a given symbol in the ***.dynsym*** section, thereby avoiding a linear search of the symbol table. A given symbol can typically be located in one or two tries through the use of the hash table.

The ***.plt*** section contains the jump table that is used when we call functions in the shared library. By default the ***.plt*** entries are all initialized by the linker not to point to the correct target functions, but instead to point to the dynamic loader itself. Thus, the first time you call any given function, the dynamic loader looks up the function and fixes the target of the ***.plt*** so that the next time this ***.plt*** slot is used we call the correct function. After making this change, the dynamic loader calls the function itself.

The ***.dynamic*** section contains some shorthand notes used by the dynamic loader.

If the ELF is converted to a shared library, the dlopen() function can be used to dynamically load a shared library into the user's memory, and you are then able to call the dynamic loader to find symbols within this shared library—in other words, you can call functions that are defined in these modules. In addition, the dynamic loader is used to resolve any undefined symbols within the module itself.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
01-01-2011

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Symbol Tables and Branch Tables
Linking Applications Together

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Handler, Louis, M.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
WBS 439432.04.07.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER
E-17552

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITOR'S ACRONYM(S)
NASA

11. SPONSORING/MONITORING REPORT NUMBER
NASA/TM-2011-216948

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Categories: 61 and 62
Available electronically at http://gltrs.grc.nasa.gov
This publication is available from the NASA Center for AeroSpace Information, 443-757-5802

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This document explores the computer techniques used to execute software whose parts are compiled and linked separately. The computer techniques include using a branch table or indirect address table to connect the parts. Methods of storing the information in data structures are discussed as well as differences between C and C++.

15. SUBJECT TERMS
Compilers; C++ (programming language); C (programming language); Computer techniques; Data structures

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STI Help Desk (email:help@sti.nasa.gov)
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	14	**19b. TELEPHONE NUMBER** *(include area code)* 443-757-5802