BR Toolset

Binary Relocatable Utilities for RTE

14Jan00

Prepared for InterWorks 2000
by Alan Tibbetts

Binary Relocatables on the HP1000

There is a secret weapon that is part of program development on the HP1000. That secret weapon is the binary relocatable file format. The binary relocatable file (B.R. for the rest of this paper) is an intermediate step between source files and object files. Although there are other operating systems which use an intermediate format similar to RTE B.R. files, they do not use them in such a prominent and useful way in the software development process as RTE does. This paper will discuss the features of B.R. files which are important to the application programmer and how to use some utilities to make the most of this secret weapon.

Some History

In the beginning, when computers had vacuum tubes and giants walked the Earth (Von Neumann, Turing, et al), computers were programmed by entering the binary values for the instructions and data directly into computer memory. As time passed, it became obvious that human beings are not that good at handling massive amounts of ones and zeroes in the raw, so the first assemblers were invented to allow the programmer to work with a more readable form of input. Although the terse and cryptic mnemonics of assembly language will never be mistaken for real human language, they are a vast improvement over binary, octal, or hex codes. Later still, compilers were invented to make programs look more like human language, starting with FORTRAN, COBOL, ALGOL and the like. The trend of abstraction of the languages continued through generations of computers and operating systems. As we enter the first year of the twenty-first century, we now have numerous so-called high level languages, some of which are actually used for useful work rather than existing simply as laboratory curiosities.

In the biological sciences, there is a saying that ontogeny recapitulates phylogeny. It seems to me that there is a similar effect in computers. Each new computer which is introduced seems to repeat the developmental steps of the history of computing. The HP1000 certainly conformed to that pattern. At the time of its introduction, it had only the most rudimentary of software systems. It was programmed in Assembler which directly generated machine codes for the program to be run. There was no mass storage, and therefore no file system. The human programmer was expected to take care of the details of where in memory the program would run and where the data would be stored. If more than one person worked on a program, they had to assign physical addresses where subroutines would be stored and linked into the mainline code.

Although this was barely adequate for the 4K words of memory that the HP2116 had at its introduction, it became too cumbersome to use manual methods when folks started working with greater amounts of memory. By the time of the HP2100A, most machines were sold with 16K of memory, and quite a few with 32K, the maximum the machine could address! It was obvious by this time that programming was too complicated to have each project team invent its own methods for talking to I/O devices, and the first Operating Systems were invented. The first O.S. for the HP1000, the SIO System, used modules which were at fixed addresses in memory. The problems of trying to assign fixed addresses that would be acceptable for a wide range of customer configurations were almost insurmountable. A partial solution for the problem was to make the sources to the HP provided code available to the end users. From a support person's point of view, this is far from ideal. A solution was sought that would allow HP to ship binary

code, but in a format that could be assigned (“relocated”) to any address in memory. The solution took the form of the B.R. format.

The B.R. Format in a Nutshell

The concept is really simple to explain. One of the more important tasks of a compiler is to figure out where everything (code and data) is going to be at run time and then form instructions that will perform the desired actions. When the programmer writes “LDA FOO” or “Status = Foo + Bar”, it has to be changed into something very, very specific for the computer to execute. The “LDA FOO” must be transformed into “LDA from location 14302" for example. Remember, the computer is only a very high speed idiot aspiring to rise to the status of imbecile someday. The conflicting requirements that the location must be precisely known, but we have no idea where the end user is going to put the code means that there is no simple answer. The complicated answer was to alter the assembler and the compiler so that they emit machine codes just as before, but with the origin of the code set at zero for every module and with each instruction which contained an address flagged for later alteration by a new step in the process that was called “loading”. We now call the process “linking”. Problem number one solved; we can put our code anywhere in memory. Problem number two: if the code is movable, how do we know how to call one module from another? The solution was to hand the problem off to the loader as well. The compiler was made intelligent enough that calls to subroutines which were not coded in the source file it was working on were assumed to be somewhere else, external to the current source, so it made a special notation in the B.R. file that indicates that the module needs some other module of a given name. The compiler also published the names of each of the subroutines in the current module to the B.R. file so that those subroutines would be available to satisfy the requests from other modules. Thus the compiler made available the entry points and externals so that the linker could weave together the separate modules into an executable program.

More Details

Programs are not composed of just programs and subroutines. There are also data structures, common, and other things that you might want to reference from a separate module. The B.R. format allows for references to these entities also. The package of code that I have been referring to as a module is also called a compilation unit. At the first statement of a program, function, subroutine, or block common, the Fortran compiler emits a B.R. NAM record. The END statement in Fortran causes it to finish up the compilation process and emit a B.R. END record. Whatever came between the NAM and END records was the compilation unit. The same structure is built by the other high level languages on the HP1000, and must be built manually by the assembly language programmer.

A unique feature of the HP1000 is the microcoded instructions which replace subroutines in the relocatable library. We refer to these entry points as RPLs and they are used by the linker to “educate” the code so that it does in-line instructions instead of invoking the subroutines of the same name.

The B.R. format also allows for various directives to be passed on to the linker to tell it such things as program priority, system common access, security access levels and so on.

Renovation and Renewal

The original B.R. format restricted the length of external names to 5 characters. When the HP2100 was supplanted by the 21MX which allowed a much larger address space, limitations that were acceptable in the old days became too confining for more modern coding styles. RTE-4 introduced extended memory addressing and laid the groundwork for the virtual memory addressing of RTE-6. Obviously, the B.R. format had to change to accommodate these constructs. While they were at it, the re-designers of the B.R. format added features that allowed even more information to be carried from the sources through to the linker, such as the file name of the source file for each module, and date stamps to show when a given module was compiled. They also added INFO records to pass information needed by the Symbolic Debug program.

Prepared for the Future

The update to the B.R. format defined some more features that were regrettably never implemented. If there had been funding to do it, the compilers could have been enhanced to pass data type information in each entry or external record to allow the linker to do type checking even for languages such as Fortran that do not do compile time type checking. As is true of most of RTE software, the good folks at Hewlett-Packard have documented the B.R. format to the end user. This has been handy on several occasions, allowing me to create a set of tools that let me tap into the wealth of information that the B.R. format makes available.

The Tools of Your Trade

The tools that I will discuss in the paper are:

NAMBR Display module information in a library

SRCBR Search B.R. files

GETBR Copy a module from within a B.R. library to a file

DELBR Delete a module from a B.R. library

MERGE Create (B.R.) library files

LINDX Create indexed B.R. library files

IMREL Inverse macro assembler

NAMBR

NAMBR is used to display information about the modules contained in a B.R. file. The NAM record for each module is displayed followed by listings of the entry point and external reference records. For HP supplied code, the NAM record contains the part number, the revision code, and a date stamp when the module was last edited. Sometimes there is a short description field as well. It is unfortunate, but most of the code written outside HP did not put as much information in the NAM records. NAMBR is used when you want an overview of a library. The runstring for NAMBR is:

NAMBR MASK

where MASK is a file mask specifying the file or files to be displayed.

SRCBR

SRCBR is used to search B.R. files for a given module name, entry point, or external reference. It is the ideal tool to use when you cannot remember which library contains the undefined entry point that the linker is complaining about. Of course you could attempt to find the missing name in the RTE manuals, assuming that you have a current copy handy, but why not search the libraries on the machine you are using?

SRCBR was invented to assist in Y2K remediation efforts. I wanted to search for all references to GetTime in a large customer application. I thought of using GREP to search the sources, but decided against it for several reasons. First, the DG program from the contributed library is faster than GREP so it would have been the tool of choice. Second, I was not sure that I had been given a complete set of sources. In the case where I am wanting to find which HP library contains a missing routine, I almost never have access to sources. Third, the customers sources each had a comment header blocks at the start of each file. That is good. Most of the comment headers were created by copying a template file, so they often claimed to reference routines that were not actually used. That is bad. By using SRCBR to find only those files that actually invoked the GetTime subroutine, I did not waste time examining files with misleading comments.

The invocation for SRCBR is:

SrcBR Type Name Mask <990311.0015>
Type = "MO" to search on module name
= "EN" to search on entry point name
= "EX" to search on external reference name
Name = name of module, entry point, or ext. ref. to find
Mask = file mask specifying files to be searched.
":::5" will be appended to the mask for you.

GETBR

GETBR is used to extract a module from a library. I have used it when I wanted to use IMREL to recreate a pseudo-source for a customer module for which the true source was no longer available. I have also used it to extract the set of HP subroutines that are referenced by one of my programs to create a merged file that will load without depending upon the libraries at a customer site. For example, I used it to allow me to load some utilities that I have written using the HpZ routines onto an RTE-4B system. Of course this technique won't always work because some of the modules in the HP library are very O.S. revision dependant, but most of the HpZ routines will run even in an RTE-2 system, as long as the CPU has X and Y registers. The invocation for GETBR is:

GetBR Lib_FileName Module <990222.2331>
Lib_FileName is the name of the library to be searched,
/libraries/$BigLb.Lib for example
Module is the name of a module to be found
and then extracted into a file named "module.rel"

DELBR

DELBR is used to create a copy of a library, leaving out one of the modules. Usually this is done so that a more recent, hopefully bug free, version of the module can be appended using MERGE. The invocation for DELBR is:

DelBR Lib_FileName Module Out_FileName <970801.1706>
Lib_FileName is the name of the library to be searched,
"/libraries/$BigLb.Lib" for example
Module is the name of a module to be found
and then deleted from the file
Out_FileName is the name of the output file to create

IMREL

IMREL is the Contributed System Library program that creates a pseudo-source for a given module. It is an accurate source because it can be run through MACRO/1000 to create a relocatable file that is identical to the one that you started with, but of course all comments are missing. I have actually read authors that have claimed that comments should not be needed in a well written program with well chosen names for variables. If you agree with them, you still won't like the output from IMREL, because all of the labels and names of internal variables are machine generated and convey no information other than the line number where the are defined. Turning IMREL output back into a reasonable MACRO source takes a good bit of patience and a lot of experience. If the original source was Fortran, the challenge is even greater, so IMREL is used only when all other methods to find the sources have failed.

MERGE and LINDX

MERGE and LINDX are the standard HP supplied programs for concatenating and indexing libraries. You are probably already familiar with their operation and the documentation is in the HP manuals, so these programs will not be described in further detail here.

Program Availability

The sources for BR Tools are available in the RTE CSL or by e-mail from the author.