Assorted Utilities: Part II

BY SAM GOLOB

Anyone who has been in the IBM mainframe systems programming field for a long time has undoubtedly noticed some trends that may not be as apparent to newer systems programmers. First and foremost is that great skill in assembler programming is apparently "no longer required." This is not to say that skill in assembler programming is no longer valuable — there's nothing farther from the truth. An understanding of the inner workings of the system is worth its weight in platinum. Rather, the situation presents itself to us in the form of a statement: "They don't grow 'em like they used to!" In other words, newer systems programmers aren't nearly as skilled in the nitty-gritty of assembler instructions and system control blocks as the old-timers were.

Having broken into the field a bit more recently, my friend Rick Fochtman explained to me why this was true. He told me that in the late '60s and early '70s, the OS/360 days, there were few convenient exit points and system facilities for getting performance information or for doing anything else you wanted. These were also the days before PL/S coding. The only way you could make the system do something it didn't do was to look into the system source code, or disassemble system object code, and code the modification yourself in assembler language. You literally had to have the same skills as the IBM code developers.

Very often, in those times, the programmer didn't have access to the layout of the IBM control blocks. In many cases, the control block layouts were in the private macro libraries, and not everyone could get to see them. So a systems programmer making a modification to an IBM module literally had to look in the module, figure out how the control block got loaded or referenced, and deduce the layout without further help. All the IBM coding was done in pure assembler language then, and someone who coded a modification would acquire a very large understanding of the workings of the original module(s), without outside help. This was a "very good school" for learning. Later systems programmers, who didn't have the opportunity to go to this "school," by-and-large didn't become as skilled as the "older" programmers.

In other words, newer systems programmers aren't nearly as skilled in the nitty-gritty of assembler instructions and system control blocks as the old-timers were.

Somewhat later, by the early '80s, things were very different. By then IBM had provided some official "exit points" where you could insert user code to modify parts of the system. Also, large numbers of pre-existing modifications to the system were at that time already available, having been coded by the old-timers. These could be merely re-fitted to the later versions of the operating system modules — they didn't have to be researched and coded from scratch. The MVS operating system was, by now, more mature. A lot more frills and tools had already been added. Large collections of user-written code, such as the CBT MVS Tape and other available sources of free code, were now readily accessible to practitioners. Modification, as opposed to coding from scratch, became more the rule of the day.

This situation led to the newer systems programmers being less skilled in assembler programming than their earlier counterparts. The "school" wasn't as good. It's even worse nowadays, with the "vendorization" of a lot of system tools. It might seem that today's systems programmers don't have to code anything. The vendors do it all for them. There's hardly any "school" at all, at least for coding's sake. To that, I'll answer one statement: We systems programmers each have to "make our own school."

WHAT WE CAN DO FOR OURSELVES

We have help, if we know where to look for it. Dave Alcock, who just submitted a 60,000-line pds of his own code for inclusion into the CBT Tape, told me that he learned to program in assembler mostly from looking at other people's code on the CBT Tape. NaSPA President, Scott Sherer, told me the exact same thing, and I, too, can also say that I learned many of my coding skills from looking at code on the CBT Tape. However, the three of us are not alone. Many of our contemporaries have learned enough to make substantial contributions to this field just from studying other people's earlier contributions.

This month I present a small sample of such code from various contributors to the CBT Tape. Some of these samples are from the old-timers. But many of them are from people like you and me, professionals who took it upon themselves to learn assembler coding skills by studying other people's clever creations.

AN EXAMPLE: SYS1.BRODCAST RESEARCH

Jim Marshall, who worked at various computer installations for the United States

Air Force and is now a civilian, has pioneered the popularization of user-written code among systems programmers. Jim had the opportunity of working with many genuine old-timers in this field. Even though Jim can be considered an old-timer like the others, his contribution is far greater. He took it upon himself to collect other people's software creations and to send them to the CBT Tape, the SHARE MVS Tape (now part of the CBT Overflow Tape) and to other tape collections. I got my own start by looking at Jim's software collections. This brings me to the current topic.

At my previous site, we administered about 11 separate MVS systems. Two of the older systems displayed a "SYS1.BRODCAST FULL" message. This was before we put in individual TSO userlog datasets, and I needed some tools to investigate. I didn't want to risk doing an ACCOUNT SYNC, which would wipe out all of the messages — some of them might be useful. As a large service bureau, we couldn't afford to offend any users. Using an updated CBT Tape program called BRODSCAN (File 234) that was submitted by Jim Marshall, I was able to see which users had outstanding messages, but I wasn't able to display or delete the messages. I then fired up some code from Tim Vanderwall that displayed the messages for each user (File 141). This was an improvement. However, there still remained the problem of deleting messages so that space on the SYS1.BRODCAST dataset could be freed. These messages belonged to users who had left the installation, and I couldn't logon to their TSO accounts without their passwords, not being the RACF administrator.

The advantage of coding a complicated system patch in ASMTOZAP format is that by changing a few ORGs, you can usually refit the patch to a later version of the IBM module. All VER and REP displacements have been calculated by the assembler.

Jim Marshall sent me a hint. He told me that the LISTBC program from IBM, which displays and deletes mail messages to each user when you logon to TSO, only finds out the user ID by looking at the PSCBUSER field in the PSCB control block. My first attempt at solving this problem was to zap core in my address space, alter my PSCBUSER field to another ID, run LISTBC to display and delete the other user's messages, and zap core again to change the PSCBUSER field back to my own ID. This worked but was awkward. Next, I wrote an authorized TSO command to do the same thing. So far, so good.

Then, my boss got wind of the fact that my program dynamically altered a TSO user's ID. He didn't like the code for such a program to be in an accessible place in the shop, so I had to rewrite the program. Therefore, I had to learn more about the internals of the SYS1.BRODCAST dataset. To make a long story short, I was able to write two other programs to display and delete someone else's BRODCAST messages by using principles I learned from looking at the code in the BRODSCAN program and from browsing the SYS1.BRODCAST dataset with the REVIEW program (from Bill Godfrey and Greg Price) on File 134 of the CBT Tape. I also wrote several programs to display but not delete any user's messages.

Why use REVIEW and not ISPF BROWSE to look at the SYS1.BRODCAST dataset? It turns out that SYS1.BRODCAST is a keyed dataset with KEYLEN=1 and LRECL=129. ISPF BROWSE uses QSAM, and cannot differentiate between the key and the data so it only shows the first 129 bytes, including the leading key, and drops the last byte of the data. REVIEW uses BSAM, and it shows all the key and data bytes. You see why you need another tool? It turns out that the last byte in a SYS1.BRODCAST record is crucial to understanding the dataset structure because for a chained message record, the last three bytes point to the location of the next record in the chain. With ISPF BROWSE, you can't see the whole pointer. How did I find this out in the first place? By looking at SYS1.BRODCAST with the UCLA Fullscreen ZAP tool (also from File 134 of the CBT Tape). Fullscreen ZAP showed the whole physical record, and I saw that ISPF BROWSE was dropping the last byte of it. All these investigations resulted in a package of programs that I wrote to handle SYS1.BRODCAST problems. I resubmitted them to the CBT Tape for everyone else's benefit (File 247). See how one contribution leads to another?

SOME OTHER EXAMPLES

Another crucial contribution, which led to other contributions, is the dynamic allocation interface from Pat MacDonald of the University of Manitoba. Pat is an old-timer in the field who has now retired from the University. Pat's contribution was to simplify programming for packages that do multiple dynamic allocations. Dynamically allocating

TOOLS FOR TECHIES

The CBT MVS Utilities Tape is a huge independently produced collection of system software tools that was started in 1975 by Arnold Casinghino and has evolved through 417 versions (Arnie did 321 of them). Additionally, a CBT Overflow Tape containing numerous other software contributions is available. These collections can be obtained from NaSPA (see page 36). Version 416 of the CBT Tape is available on the NaSPA CD-ROM Volume 1, Version 4, the NaSCOM Internet server, or via FTP at ftp://www.naspa.net/cbt416.

datasets in a program, using SVC 99, is not especially difficult to code, but the set up is lengthy, and it would be very helpful to simplify the process. Pat's package, which can be found on Files 089 and 090 of the CBT Tape, has been employed by many other programs on the tape, notably the ARCHIVER package from Rick Fochtman, which is on File 147.

Rick's ARCHIVER package is unlike any that I have ever seen or even heard of. I don't even think there is a vendor product that does what the ARCHIVER does. The ARCHIVER will take any normal non-VSAM system programmer data — source modules in pds or sequential form, load modules, PSF objects, or whatever — and load it into one archive that is a VSAM dataset in highly compressed form. You can fully recover the data, or pds, member by member, from the VSAM archive. Additionally, the ARCHIVER has a facility to dump the "members" of the archive into a tape dataset, allowing you to recover all the members from the tape as well. The main difference between the VSAM archive and the tape archive is that you can add or delete members with the VSAM archive but you can't do so from the tape archive.

Compression in the ARCHIVER is enormous. Version 5.0 of the ARCHIVER uses the Huffman Tree algorithm for compression. Using Version 5.0 of the ARCHIVER, I once dumped approximately 178 sizable pdses onto one 3480 non-compressed cartridge. At a later consulting job that tape proved useful. The ARCHIVER is on File 147 of the CBT Tape. Rick Fochtman has been working on further improvements to it.

ASMTOZAP was written by Howard Gilbert at Yale University, another old-timer. ASMTOZAP is a PL/I program that allows you to code a system modification zap as assembler code. You code the zap in assembler language using special control cards starting with the keyword *ZAP. Then you assemble the ASMTOZAP-format patch with your assembler. The *ZAP keywords come out as assembler comments in the listing. Finally, you post-process the assembler listing that contains machine instructions and location counter information with the ASMTOZAP program to generate a system zap to the original module. Some examples of ASMTOZAP patches can be found in File 369 of the CBT Tape by looking at members starting with the letters "LM" (for Local Mod). My July 1989 column (File 120 of the CBT Tape) also illustrates ASMTOZAP processing.

The ASMTOZAP program looks at the *ZAP keywords in your assembler listing and performs actions to generate card images. If you want ASMTOZAP to punch a card, such as a ++USERMOD card, you code a *ZAP CARD statement in your assembler code. To make ASMTOZAP start generating VER statements from the assembler code to follow, you can code *ZAP START VER. ASMTOZAP will generate REP statements after *ZAP START REP has been coded. Finally, *ZAP END tells ASMTOZAP to stop generating cards after that point in the assembly listing. Displacements into system code are indicated by ORG statements in your assembler code. The advantage of coding a complicated system patch in ASMTOZAP format is that by changing a few ORGs, you can usually refit the patch to a later version of the IBM module. All VER and REP displacements have been calculated by the assembler.

ASMTOZAP was written in PL/I Optimizer. Some shops aren't licensed for PL/I. However, I modified a version of ASMTOZAP so it would run under the free PL/I F compiler and library (File 092 of the CBT Tape). ASMTOZAP, and its PL/I F equivalent ASMTOZAF are available on File 044 of the CBT Tape. Load modules can be found on File 035. Any MVS shop can now run ASMTOZAP. Take a look at ASMTOZAP. It may make life much easier for you if you have system mods.

This month's tool discussion doesn't even scratch the surface of what's available. For more information on the CBT tapes and their documentation, visit http://members.aol.com/cbttape. Good luck. See you next month. ts

---

NaSPA member Sam Golob is a senior systems programmer. Sam can be reached at sbgolob@aol.com or sbgolob@ibm.net.