Washington Systems Center
Gaithersburg, Maryland
Technical Bulletin

# VM/370 MAINTENANCE MADE SIMPLE

Written by: R. W. Benham
Edited by: A. A. Gigliotti
G. E. Hollendursky

GG22-9277-00
May 1982

## PREFACE

There seem to be many misconceptions regarding the generation and maintenance of VM/CMS systems. I believe that simplicity has always been the hallmark of VM/370. Many of the problems arise from new users imagining that the processes must contain a certain complexity which simply is not there.

Much of the imagined complexity has doubtless been caused by the many procedures and EXEC's which have been developed over the years in an effort to simplify the generation and maintenance process. And while they do, indeed, provide valuable and time-saving tools to the initiated, they tend to obscure the basic processes from the struggling novice.

The purpose of this paper is to try to clarify the process.

## CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

VM/370 is comprised of a number of components. The two most significant of these are CP (the Control Program) and CMS (the Conversational Monitor System). These are the only two components with which we will deal at any length, as they are the head and heart of the system and are the only components which are absolutely necessary to get the job (generation and maintenance) done.

Now let's look at CP. CP has but one real mission in life, and that is to multiprogram virtual machines. True, it provides other nice functions as well, such as SPOOLing, and VMCF communications between virtual machines, and DIAGNOSE interfaces to enhance the function and performance of virtual machines, but the fact remains that CP really exists to create and juggle a whole bunch of IBM 360 and 370 virtual computers on one real computer.

And how hard is that? It may sound complex at first. If we think about it, it really becomes (conceptually, at least) quite simple.

The most difficult part of any system design or programming problem is always definition, followed a close second by change ("That's not what I meant!!" says the user to the programmer). CP's "problem definition" is almost cast in concrete. It may be found in IBM publication GA22-7000, "IBM System/370 Principles of Operations". It's the function described therein that CP makes every attempt to provide to a multitude of concurrent users.

But what does a user do with his own IBM 360/370 computer? Surely not everyone wants to be a computer operator or a systems programmer, playing with PSW's and general registers and control registers and all that neat stuff. No, generally they will want to run some operating system which will provide the services that we have come to expect of operating systems, such as I/O access method support, and compiler support, and support of various end-user programs like STAIRS and QBE and SPF, and storage management, and device management, and on and on.

Now the user might choose to run any number of operating systems in that virtual IBM 360 or 370 that CP has created. Over the years, IBM has developed quite a number of such systems - BPS, TOS, BOS, DOS, PCP, MFT-I, MFT-II, OS/VS1, DOS/VSE, SVS, MVS... Since each of these systems was designed to run in an IBM 360 or 370, each of them will most probably run in that virtual machine that CP has created.

But wouldn't it be nice to have an operating system that was custom designed to run to maximum advantage in a virtual machine? One that would provide all that function that we have come to expect of an operating system?

Enter our second component - CMS. CMS is just such an operating system with all of those above mentioned functions, and while it would be a gross misrepresentation to infer that there is no connection between CP and CMS, for our purposes let us just say that CMS is just another operating system running in a virtual machine. So far as CP is concerned, it might be DOS or BPS or whatever. CP is just managing the virtual hardware and cares not what we run in it.

Now what is the point of this discussion about CP, and virtual IBM 360's and 370's, and operating systems? The point is to dispell the myth that some virtual machines have special qualities and do things differently from garden variety virtual machines. The fact is that CP creates and manages all virtual IBM 360 and 370 computers as they are defined in the current system DIRECTORY. All virtual machines are created equal in CP's eyes, and the fact that one is called MAINT and one is called RSCS is of absolutely no consequence to CP.

So let's talk now of convention.

As a matter of convenience and in an attempt at standardization, a virtual machine with the name of MAINT has become something of standard in VM/370 installations. But this seems also to have led to the belief that this virtual machine has certain qualities that other machines like CMS1 and CMS2 do not. This is sheer nonsense.

But surely, you say, there is something unique about MAINT? And yes, I answer, but the only uniqueness lies in the fact that the MAINT virtual machine is generally defined in the system DIRECTORY as having more privileges than most virtual machines are given (via the privilege class entries in the USER statement), and as having write access to more system related mini-disks and full-disk volumes than the "average" userid.

To look a bit more closely, the userid generally named MAINT is ordinarily given privilege classes ABCDEFG which allows that particular userid to use any command in the system (including SHUTDOWN, so be careful). It is also generally defined as a rather large machine - typically 3 Megs with the option of redefining itself right up to the 16 Meg limit. It will require this large address space to install Discontiguous Shared Segments which typically reside above the "average" user's address space. It will generally have MDISK statements in its DIRECTORY entry for all of the system-related mini-disks (the CMS system disk, the disks used for CP and CMS maintenance, etc). Other MDISK statements give it read/write access to full volumes so that it may do systems maintenance such as re-writing the CP nucleus, changing disk allocations, and writing the DIRECTORY itself.

Those are the things which make MAINT "different" from an "average" user.

THIS PAGE INTENTIONALLY LEFT BLANK

All VM/370 installations will have at least one CP-Owned volume. The volume on which the CP nucleus resides must belong to CP and cannot be removed while the system is running.

Most VM/370 installations will have more than one CP-Owned volume. Any volume that is to be used for CP paging, SPOOLing, or T-Disk space must be defined as a CP-Owned volume in DMKSYS. One should keep in mind, however, that the DASD space used for CP functions (paging, SPOOLing, T-Disk, DIRECTORY, SYSWRM, SYSERR, SYSCKP, etc.) is NOT affected by DIRECTORY entries. The SIPO/E and the clever user may specify such areas under NOLOG entries in the DIRECTORY for documentation purposes but there is NO checking done by CP to avoid overlaps.

Let's look at some of those areas and see exactly where they are defined for use.

Each CP-Owned volume must be formatted and allocated by the CP FORMAT/ALLOCATE program. This is quite different from the CMS FORMAT program since CP and CMS are two very different "operating systems", and use different disk formats.

The FORMAT function of the program formats the entire disk (or mini-disk) to 4096-byte blocks. The first cylinder (or 16 blocks with FBA devices) contains a volume label, an IPL record, a dummy VTOC, and an allocation bit map.The allocation bit map defines how the space on the volume is to be used.

The ALLOCATE function of the program fills in the allocation bit map according to the user's specifications (PERM, TDSK, DRCT, TEMP, PAGE). If the ALLOCATE function is not run, the default from the FORMAT function is PERM for the entire disk (or mini-disk).

When space is allocated as anything but PERM (i.e. TEMP, TDSK, PAGE, DRCT), that space has been given to CP and if the same space is also inadvertantly given to some user via a MDISK statement with write access in the DIRECTORY you can expect disaster. Again, no checking is automatically done by CP, it is quite possible to allocate space on a volume as TEMP (i.e. for CP paging and SPOOLing) and then create a USER entry in the DIRECTORY with the same cylinders (or blocks) allocated as a user mini-disk. The results will be that CP paging/SPOOLing occurs in the same area that a virtual machine is writing. No one checks but YOU!

So, simply stated, keep track of how you have allocated your packs with the FORMAT/ALLOCATE program. Place USER NOLOG entries in the DIRECTORY to document all your allocation in one DIRECTORY. Remember that those DIRECTORY entries for TDSK, TEMP, PAGE, DRCT, and other CP-owned areas are for your convenience only, and are of no significance to CP.

But is the space allocated as PERM there to use at will for mini-disks? No, at least not all of it. In DMKSYS and in DMKSNT you define areas for CP for the CP-nucleus, various error recovery areas, and areas to be used for the storing of SAVED SYSTEMS and Discontiguous Shared Segments. You have explicitely told CP where and how much DASD space to use for these various functions, and CP does NOT go to the DIRECTORY to find out where these areas are. The DIRECTORY has only one function, and that is to describe to CP just what virtual IBM 360/370 computer to create when a user logs on to the system. So, again, it is up to YOU to assure that there are no overlaps.

In summary, be aware that there are really three different things to look at when trying to determine how your DASD space is allocated: first, how the real volume was allocated with the FORMAT/ALLOCATE program; second, what areas were given to CP in the DMKSYS and DMKSNT ASSEMBLE files; third, how is mini-disk space allocated in the system DIRECTORY. The second and third are easy, in that you need only look at the files, and perhaps do some simple calculations to convert pages to tracks or blocks. The first - if you have lost all record of how the disk was allocated - can be determined by running the FORMAT/ALLOCATE program and simply specifying END when prompted for the new allocations. The program will respond with the current allocations for the disk. Then write it down (or capture it in a file) and save it.

We will now turn to the MAINT virtual machine and look at certain of the mini-disks customarily associated with this service machine.

Once again the reader should be aware that the addresses used for these mini-disks are a matter of convention and not hard and fast rules. In fact even within IBM's own procedures, the conventions are periodically changed. VM/SP used 293 vs 193 for the SIPO/E.

Let's lock at what we really need in order to maintain our system, and then explore the logic behind the choice of where to put things.

First, consider that we are really maintaining two systems, CP and CMS, which, though similiar in maintenance procedure, are quite totally different in function. Therefore it becomes convenient (although not absolutely necessary) to logically and physically separate the various pieces of the two systems. The logical separation is handled quite nicely by a simple naming convention - a unique three-character prefix is used for each CP or CMS module to identify the component to which it belongs. DMKxxx identifies a CP module, DMSxxx identifies a CMS module. The three characters following the prefix give some indication of the function of the module - DMKSCH is a CP scheduler module, DMSACC is the CMS ACCESS command module, etc.

Now let us digress a moment and review the VM/370 service philosophy.

All of CP and CMS is programmed in assembler language. When a new release level of VM/370 appears the assembler source is re-sequenced to include all updates which have been incorporated during the life of the previous release. This becomes the base source and, with a few exceptions, that source does not change throughout the life of the release. The current release level of VM/370 is release 6, the current release level of VM/SP is 1.1 (that 1.1 is simply a mid-life merging of PTF's to source).

As fixes to problems become available, they are provided to customers via PUT (Program Update Tapes) or PLC (Program Level Change) tapes in the form of UPDATES to the base source. If that was all that was provided, it would require that each customer re-assemble (with the provided updates) every module in the system that had changed. As this is obviously not feasable, IBM also ships the TEXT decks which result from the re-assemble of those modules. And, as many of the CMS commands are MODULES created from TEXT decks with the LOAD, INCLUDE, and GENMOD commands, those MODULES are also shipped in order to reduce the customer effort to install updates.

A few other things are generally included in a PUT or PLC. They include updated HELP screens, EXEC's, MACLIB's, and occasionally a replacement for a source module - ASSEMBLE file - which may have been split into two modules or changed in a way inconsistent with a simple update file.

Some users may not keep CP or CMS source (ASSEMBLE) files online, as it is rarely used - unless your system is heavily modified. Since IBM ships all TEXT decks which have been modified, the source is only needed if customer modifications are required or if a problem is identified which requires an on-site modification. In this case, the required source program can be retrieved from the source tapes which are supplied by PID with the VM/370 system. All of the DMK (CP) modules are on one tape, and the DMS (CMS) modules on a second tape volume. It would not be correct, however, to infer that online source is a luxury. Much time may be saved, and the very best system documentation (the source itself) becomes immediately available when the source is kept online.

So we have reduced our requirements for maintaining a CP/CMS system to CP TEXT, MACLIB, EXEC, and UPDATE files, and CMS TEXT, MACLIB, EXEC, MODULE, HELP screens, and UPDATE files.

By convention, (which changes from time to time) these files are placed on four mini-disks which (also by convention) belong to the user MAINT. They are - 190, 193 (or 293), 194, and 294. The SYSGEN manual also recommends a 201 disk for containing the CPEREP TXTLIB's, which have generally been kept on the 190 disk.

Let's press on then. We really have only five mini-disks to contend with, so let's look at them and see what they contain.

## MAINT's 190 disk

This is historically the CMS System disk, and as such is a bit different from the other mini-disks. A system disk (be it for a "real" system or a "virtual" system) needs space for writing the system nucleus so that the system can be IPL'ed. When a disk is formatted using the CMS FORMAT command, the whole mini-disk is normally formatted for CMS use and as such the entire mini-disk would be available for writing CMS files. This would have a most adverse effect on the operation of the system if a file were written over the CMS nucleus (which is NOT a CMS file, but an IPL'able operating system). In order to prevent such an occurance, the CMS FORMAT command has a RECOMP option which makes available only a specified portion of the disk for CMS file use. You can see this difference by comparing the results of a CP QUERY VIRTUAL 190 command which will show the true size of the mini-disk, with a CMS QUERY DISK S command which will show the size of the disk in terms of what the CMS file system has available for use. They had best be different. This disk (190) contains everything that is necessary to the generating and running of CMS. It contains TEXT files, EXEC's, MACLIB's, and MODULE's, as well as the HELP screens for a BSEPP, SEPP, or VM/SP system.

This is the disk to which all of your CMS users will have read-only LINK's, in order that they may IPL and use CMS. When CMS is IPL'ed it ACCESSes the 190 (IPL) disk as the S-Disk and, as whenever a CMS disk is ACCESSed, builds an in-storage directory to the files on the disk. In an effort to reduce the size of that in-storage directory, however, only filemode number 2 (S2) files are included in that directory. Thus, if you wish to be able to "see" all of the S-Disk files, you must also access it as some other mode letter - e.g. ACCESS 190 C.

## MAINT's 191 disk

This is simply the MAINT machines work disk, and is generally used to contain those ASSEMBLE files which are needed for day-to-day systems work (DMKRIO, DMKSYS, DMKSNT). It may also contain the system DIRECTORY, unless you choose to hide that away in some obscure spot for reasons of security.

## MAINT's 193 (or 293) disk

This disk contains the UPDATE and AUX files for the CMS system. These files (and this disk) are needed only if a local modification is to be made to CMS source. It is handy to have around, however, because you can find out from the FILETYPES on this disk whether or not a particular PTF is on your system. Let's say you call the Support Center with a CMS problem, and they ask if you have APAR number 12345 on your system. If you ACCESS the 193 (or 293) disk as (say) your B-disk and issue LISTFILE * S12345DS B, you should get a hit if that PTF is on your system. FLIST is even nicer, in that you can simply enter FLIST * *12345* B, and not worry about the naming conventions for system UPDATE files.

## MAINT's 194 disk

This disk contains all of the code that you normally will need to do a CP Nucleus Generation. It has all of the CP TEXT, MACLIB's, and CNTRL files necessary for putting together a CP Nucleus which we will look at more closely in a bit. This is really the CP counterpart of the 190 disk for CMS, excepting that there is no nucleus on this disk - it contains the pieces for putting together a CP nucleus for writing on the IPL'able system disk.

## MAINT's 294 disk

This disk contains the CP UPDATE and AUX files necessary for applying source changes to CP modules. It is the CP counterpart to the 193 (or 293) disk for CMS. As with the 193 disk, it is used only when it becomes necessary to apply local source changes or when one wishes to see if a particular PTF is on ones system.

And that's the lot. True, MAINT has a bunch of other disks which normally belong to that virtual machine, but all of the pieces used for normal CP/CMS maintenance (with the exception of source, which may be selectively loaded from tape as needed) are contained on those five mini-disks. And if you consider that 193 (or 293) and 294 are needed only if source changes are to be made, that cuts the list to three - 190 for CMS generation, 194 for CP generation, and 191 for a work disk if certain modules (such as DMKRIO) need to be re-assembled.

The one exception to the above, is that the MAINT machine (or whatever machine you choose to use for maintenance) must have READ/WRITE access to the system volume which is to contain the system CP Nucleus and the system DIRECTORY. This disk need not be ACCESS'ed as a CMS disk, but must be available whenever a DIRECTORY is to be written or when a new CP nucleus is created.

We shall press on now and look at exactly how a CP and/or CMS nucleus generation is accomplished.

This section will be a breeze for those among us who remember the days of stand-alone programs which were loaded from cards via a real card reader. Not JOB's loaded from cards but whole programs which consisted of a boot-strap IPL program of generally two or three cards, followed by a LOADER program, which loaded and relocated TEXT decks, resolved external references, printed a map showing where everything was loaded, and finally gave control to the program which it had loaded.

Well that's exactly how a CP or CMS nucleus is generated and loaded.

Now many EXEC's have come into being which have a way of obscuring the facts - EXEC's like GENERATE, and GENERBSE - but the key MODULE which (with a little information from us) creates those great big stand-alone card decks for loading is VMFLOAD.

The process which will now be described is, for all practical purposes, identical for CP, CMS, and for certain other sub-systems (such as RSCS) which have their own IPL'able nuclei. It is quite simply a process of piling together many card decks - certain of which we may have produced via assembly of source - with an IPL program and LOADER program in the front, dozens of TEXT decks in the middle, and an LDT (Load Terminate) card at the end which directs the LOADER to the symbolic address to be given control after all of those TEXT decks have been loaded into storage.

You could very well do all that manually - that is, identify each of the appropriate decks to punch, punch them using the CMS PUNCH command, gather up all the cards from the card punch, place them in the card reader, and IPL from the reader. Don't! For a CP nucleus generation, you would have something in excess of 200 PUNCH commands to issue and about 16 to 20 thousand cards to shuffle.

But the process that you do use is functionally identical to that described above, except that it is automated by supplied lists of the decks to be punched, and the decks are "punched" to, and IPL'ed from, disk by virtue of CP virtual SPOOL devices, thus reducing the element of human error and the dropping of cards.

Let us now turn our attention to the bits and pieces which are used to automate the process of building a nucleus. As we explore these various modules and files, keep in mind that they are the ones which are invoked when we use the various system-supplied EXEC's such as GENERATE and GENERBSE.

We really have only three things to consider (four if you are planning to include a V=R area in your CP nucleus generation).

## The CNTRL file

This file (of which there are several, those for CP normally residing on the MAINT 194 disk, and those for CMS normally on the MAINT 190 disk) is a multi-purpose file. CNTRL is a filetype which is recognized by the CMS UPDATE command and by the VMFLOAD MODULE. UPDATE is imbedded in the VMFASM EXEC when we assemble a system source module. For purposes of VMFLOAD, the CNTRL file used specifies the search order which is to be applied to filetype when punching out all those TEXT decks which will make up our CP or CMS nucleus.

This allows us a certain amount of freedom in applying local fixes or modifications, in that not all of the decks punched need have the filetype of TEXT (although they will all be in the TEXT, or object code format). Thus, we might wish to have all locally altered decks identified with a filetype of TXTLCL for instance, rather than TEXT. With a properly modified CNTRL file, we may then specify that the VMFLOAD module is first to look for decks with a filetype of TXTLCL, before searching for one with a filetype of TEXT. We can, in fact, have many levels to the search order by simply adding the appropriate entries to the CNTRL file.

As supplied with the system, there are a number of different CNTRL files from which we select depending upon our requirements. For example, with VM/SP, we have DMKSP CNTRL which is used for creating an ordinary uniprocessor version of CP, and DMKSPA CNTRL for those needing Attached-Processor support, and DMKSPM for Multi-Processor systems.

## The Loadlist EXEC file

The next file to consider, is one which contains the names (and optionally filetypes) of all the various TEXT decks which are to be punched out to make up our system.

As with the CNTRL files, many different loadlist files are provided with the system, to be used depending upon our particular needs and upon which nucleus (CP, CMS, RSCS) that we happen to be creating. They are not quite so easy to identify as CNTRL files, however, since they have a filetype of EXEC, as do countless other files which are not loadlists. Generally, they will have the characters LOAD or LD as a part of the filename (CPLOAD, CPLOADSM, CPLDBSE, etc.) and can thus be identified.

Loadlists will generally have two things in common. They will have a loader specified as the first file to be punched which will probably be named DMKLDOOE LOADER. This loader is supplied with the system as a stand-alone loader which produces a load map on a printer at address 00E - the address most commonly used for the CMS virtual printer, and a common real printer address as well. The second commonality will be the last file specified to be punched. This will normally be a file with a filename of LDT, and if we inspect files named LDT in our system, we will generally find that they consist of a single card image which is the LDT (Load Terminate) command to the loader (DMKLD00E LOADER). This notifies the loader that its work is done and requests that the loader now pass control to the program which it has just finished loading.

The dozens or hundreds of line-items which come between these first and last items (LOADER and LDT) are the filenames of the TEXT (object) decks that are to be punched out to make up our nucleus. Notice that the filetypes are not generally specified within the loadlist, since we may wish to take advantage of the flexibility offered by the CNTRL file process to affect a search order containing filetypes of other than TEXT.

## The VRSIZE MODULE

The next item to be optionally considered is a module on MAINT's 190 disk named VRSIZE. This module has a very simple job to do. If you plan to generate a CP nucleus which will allow a V=R area, you will be using one of the loadlists which accomodate V=R systems (VRLOAD, AVLOAD, etc.).

The big difference in a V=R system, is that the CP nucleus - after page zero is loaded is relocated to some place in upper memory in order to accomodate the V=R system. The V=R system runs in storage at virtual addresses which are the same as real addresses (with the exception of page zero which belongs to CP).

The simple function of the VRSIZE MODULE then, is to ask two questions. The first is whether or not you plan to have a V=R system. If your answer is NO the module goes away without doing anything. If you answer YES, the second question is asked. How big is your V=R area to be? You answer, and the VRSIZE program then produces a three record file called DMKSLC TEXT. This is an SLC (Set Location Counter) Loader command (followed by an ESD (External Symbol Dictionary) card and an END card). When encountered by the loader (DMKLD00E LOADER) these cause the loader to skip the amount of storage specified by the SLC command before resuming the loading and relocating of TEXT decks. Thus we leave a "hole" for the V=R virtual machine.

If your curiosity leads you to explore the various loadlists (CPLOAD, VRLOAD, etc.) you will find that those loadlists which are supplied for a V=R system will have an entry for DMKSLC, where the non-V=R loadlists will not.

A note for the curious. If you have become intrigued by the various loader commands (SLC, LDT etc.) mentioned here, the "CMS Command and Macro" manual has a good explanation of many of the loader-recognized commands under the LOAD command section. There are many such commands which were most familiar in the days of BPS card systems.

## The VMFLOAD MODULE

This fourth element in our nucleus generation process is a MODULE file normally found on MAINT's 190 disk (the CMS system disk) with a name of VMFLOAD. It is this module that puts together all of the above mentioned pieces.

VMFLOAD expects two operands when it is invoked. The first is the name of the loadlist to be used to identify the decks to be punched, and the second is the name of the CNTRL file to be used for determining what filetypes to look for when running through that loadlist.

Thus, the command VMFLOAD CPLOAD DMKSP would cause the VMFLOAD program to simply search for each file specified by name in the loadlist CPLOAD and append a filetype according to the search order specified in the CNTRL file DMKSP. The actual disk order of search is the standard CMS search order starting with the disk ACCESSed as the A-disk and running through the alphabet. The first file found that satisfies the search criteria is the one punched by VMFLOAD.

Notice that the VMFLOAD usage of the CNTRL file in locating filetypes for which to search is from the TOP DOWN so as to be compatable with the VMFASM EXEC's usage in naming those TEXT decks (as discussed in a later section) which is from the bottom up. The update level identifier (ULI) on the MACS record is ignored, as are any ULIs of "PTF". VMFLOAD prefixes the characters TXT to the ULI to determine the filetype for which to look. Thus, if the ULI is LCL, VMFLOAD will search for filetype TXTLCL.

## Some Notes

It should be mentioned that when the VMFLOAD command is used directly, none of the initial 'set-up' which is normally done by the GENERATE EXEC is automatically done for you. Thus, you are responsible for assuring that the necessary assemblies have been done (DMKRIO, DMKSYS, etc.) and that the appropriate TEXT (object) decks are available to be punched as a part of the nucleus to be generated. Nor is the punch SPOOLed to your virtual machine, so unless you do it, the resulting 16 thousand cards or so will be dispatched to the real system punch which is probably not what you had in mind.

But on the plus side, you are now working on a more or less basic level with the system, so that if something goes wrong, you will know where you stand.

GENERATE also takes that 16 thousand card program which has been punched, reads it back in and writes it to a tape, then rewinds the tape and IPL's the new nucleus from the tape. The theory is that you now have a copy of the nucleus (in IPL'able format) on a tape should you ever have to restore your CP nucleus in a stand-alone environment. The fact is that most installations never save or use the resulting tape anyway, so why create it? Remember that the big punch deck you have created can always be read in with some filename and saved, although the short time it takes to re-create it with VMFLOAD generally makes this unnecessary.

Another point perhaps worth mentioning is that since CP uses no "files" during operation, and is totally resident or paged (as discussed later), ALL changes to CP, even seemingly trivial ones such as adding a new FCB (Forms Control Buffer) load for a real printer, require a CP nucleus generation. With CMS, however, this is not always true, as many CMS functions are simply MODULES or TEXT files that reside on the CMS system disk and are not a part of the CMS nucleus. You can determine whether a CMS nucleus generation is required by simply scanning the CMSLOAD EXEC loadlist to see if the DMSxxx TEXT deck you are changing is part of the CMS nucleus. If not, no CMS nucleus generation is required, although you may have to re-install certain SHARED SEGMENTS as discussed later.

One other thing should be noted here. Should you desire to create your own loadlist EXEC (for a customized SMALL CP option, for example) you should start with a standard system-provided list, and comment out (an * in column one) those TEXT decks not to be included. And be SURE to check your list against new loadlists which are supplied with PUT tapes, since they have a way of changing from time to time.

## Summary

What has been described here is the general process of creating an IPL'able nucleus in an effort to demonstrate that it is basically a simple process of combining a lot of object programs into a single program which can be IPL'ed in a real or virtual machine. The process is the same for CP, CMS, and RSCS (RSCS has its own loadlist EXEC and CNTRL file which have not been discussed here). The workings of the various IPL'able programs differ, however, once they are IPL'ed, and that will be discussed subsequently.

As a sort of recapitulation of just what we have created with this VMFLOAD process let's look at that big card deck once more. The first three or four cards will be a little bootstrap loader which will initially get control when we issue an IPL to the device which contains our card deck. That little loader will serve to load the DMKLD00E LOADER which will then be given control. DMKLD00E will now continue the process by reading the card images which follow it (these are simply data to the loader), locating them in storage as they are read, and, when that LDT card is encountered the loader (DMKLD00E) will in turn give up control to the 'data' it has just loaded. Thus the IPL process has really caused the loading and execution of three 'programs' - the bootstrap loader loads the loader and gives it control, the loader loads the object decks and gives them control, and the program made of the object decks then executes. We now must look at CP and CMS separately, as they each have their own way of doing things.

# CP/CMS NUCLEI - FROM CARD TO DISK

We have at this point succeeded in putting together a whole bunch of card images from various CMS files and producing an IPL'able 'program'. What we do next is to IPL the device (probably a virtual reader) which contains that big card deck. As explained earlier, when we do this a program is loaded into storage (virtual, if we are working in a virtual machine) and given control. Up to this point the process has been identical regardless of whether we are generating a CP, CMS or RSCS nucleus. Now we must look at CP and CMS separately.

## The CP Nucleus.

The CP nucleus is self-contained, with all the information it requires supplied within object decks. Thus, it asks no questions, but simply goes about its business as directed in the user-tailored module DMKSYS. It is in this module that we tell CP where, and upon which volume to write itself. The only caveat is that we must have write access to the volume which we have specified, at the address specified, and with the volume label specified.

PLEASE NOTE. If we are doing all this in a virtual machine - as we most probably are - the device address is VIRTUAL. It may or may not be the same as any real device on the system. This seems to be an area of major confusion to new users who tend to insist upon specifying a real device address in DMKSYS but find that they get an error message from the CP nucleus program when it attempts to write itself to that disk because the MAINT machine 'sees' that device at a different virtual address. Remember, no magic, when you are working in a virtual machine, that machine has access only to those devices which are given to it via MDISK or DEDICATE statements in the DIRECTORY, or via ATTACH commands after logon, although the user can redefine the virtual addresses of those devices with the CP DEFINE command. Just because it happens to be a CP nucleus that is trying to do something does not alter this basic fact.

The key to success then is to have enough virtual storage in your virtual machine (3 Megs is generally a good number), and to have write access (this has nothing to do with CMS ACCESS, as at this moment CMS is no longer in the picture, we have IPLed that new CP nucleus in our virtual machine) to the volume and the cylinders or blocks which we have specified in DMKSYS to contain our new CP nucleus.

If all goes well, that new nucleus will write itself to the specified disk and then load a disabled PSW with a code of 12.

**VM/SP also nicely informs us just where it has written itself on the disk.**

Now, how does this affect our system - since we are probably rewriting the real CP nucleus? Not at all, until the next real system IPL. The CP nucleus is only used at initial IPL (or system restart). At that time, the 'resident' nucleus is brought into real storage and there it stays. The 'pageable' part of the nucleus is loaded and immediately paged out and from that point until the next IPL the actual CP nucleus on disk is never referenced.

As for what parts of the CP nucleus are 'resident' and what are 'paged', this is decided by one of those TEXT decks that was included in the loadlist we used to create the new nucleus. Everything up to the DMKCPE module is resident. Everything following DMKCPE is paged.

So, as you can deduce from the above, you can go ahead and regenerate your CP nucleus six times over while the system is running without affecting anything. Just hope you have a good one there the next time you IPL. I shall not insult anyone's intelligence by mentioning system backup tapes, etc...

## The CMS Nucleus

When we IPL that card version of the CMS nucleus that we created using VMFLOAD, we must be prepared to answer a few questions. Unlike CP, CMS requires no user assemblies. There is no equivalent to DMKSYS in the CMS nucleus, and therefore we will be prompted by the program for information similiar to that supplied to CP in DMKSYS (although the information required is not nearly as complex for the CMS nucleus).

We are given the opportunity to customize certain messages, which appear on terminals and printed output, to our own requirements, and we are asked for information concerning the disk addresses to be used. Where should this nucleus be written (normally 190)? What address will be used when the resulting CMS nucleus is IPL'ed from disk (also, normally 190)? What address should be used for the Y-Disk (normally 19E)? Notice that in keeping with our thesis, very little is "hard wired" in VM/370. We take certain things for granted - such as virtual address 190 for the CMS residence and virtual address 19E for the Y-Disk (the default extension to the S-Disk) - but these are but convenient standards, and we may, should we wish, have multiple CMS nuclei on separate mini-disks provided we have properly prepared the mini-disks (remember FORMAT with RECOMP?). The SIPO/E, in fact, provides a 390 mini-disk for the MAINT virtual machine for just this purpose - to contain a test CMS nucleus.

We will also be asked if we really want to re-write the CMS nucleus (one last chance to back out), and whether cylinder 0 should also be re-written with the DASD IPL record which will bootstrap the CMS IPL process to point to our nucleus area. We will answer "YES" to both questions.

NOW, one other question will be asked for which you will need a predetermined answer, and that is the absolute location (relative to the MINI-DISK, NOT to the REAL DISK) of where this nucleus is to be written. If you are not prepared with the correct answer, you may find yourself almost coming to blows with CMS, when it refuses to accept your answer. The correct answer (provided the FORMAT RECOMP was done correctly) will be the cylinder (or for FBA devices, the block multiplied by 2, 4 , or 8 depending upon the CMS blksize - 1024, 2048, or 4096 - of the system disk) which we see when we do a QUERY DISK S. With CKD dasd types (2314, 3330, 3340 3350, etc.) it will be the number which appears under the CYL heading in the response to QUERY DISK S. With an FBA device (3310, 3370), the CYL column will indicate FB, and we must take the BLK TOTAL figure (this is the number of CMS blocks) and convert it to FBA 512 byte blocks by multiplying by either 2, 4, or 8, depending on the value of the BLKSIZE column - 1024, 2048, or 4096. This value of cylinders or FBA-blocks is the number of cylinders or blocks that the CMS file-system "sees" as available for use. In fact the actual mini-disk size will be larger (as may be seen by using the CP QUERY VIRTUAL 190 command, as CP is unconcerned with the CMS file-system and concerns itself only with the "hardware") if we have properly done our FORMAT-RECOMP. Since that number - the one we arrived at with QUERY DISK S - is a TOTAL number of cylinders or blocks, it will also be the RELATIVE number (relative to zero) of the first cylinder or block NOT "seen" by the CMS file system. There is one additional caveat with FBA devices. The FBA block address specified must be a multiple of 256 as specified in the "VM SYSGEN" manual - so you would do well to have done your homework before this point.

Assuming that all of the above questions have been answered to the satisfaction of CMS, it will now write itself to the specified disk and then immediately IPL itself from that disk, so you will get immediate gratification if all went well.

## A Suggestion

Now that we have discussed all these things, go ahead and try them. So long as you don't have write access to the real CP system disk or the CMS system (190) disk, you can't hurt a thing. And if you refrain from issuing an IPL 00C to your reader after you have created a card-image nucleus, you won't change anything even if you have write access to those disks.

I would suggest that you logon to some unprivileged (class G only) machine with about 3 megs of storage. Get read access to MAINT's 194 mini-disk (you will no doubt have a read-only link to MAINT's 190 mini-disk, but you will have to ACCESS it as some other mode letter than S, in order to "see" all of the files you will need), then define a Temporary disk , do the FORMAT, and the FORMAT-RECOMP on that disk, and do some of the things we have talked about. You needn't worry about destroying anything so long as you are a class G user without write access to those system disks.

Use VMFLOAD to generate a new CP nucleus. Then IPL it and take a look at the load map that is created in the process. You will get an error message when CP tries to write itself to a non-existent disk, but you will have all the by-products of a successful gen.

Then use VMFLOAD to create a new CMS nucleus, and you can even go ahead and write it on that T-Disk, and put in your own installation headings and get a feel for the process. It will take the mystery out of it.

## On CP

As mentioned in the previous section, you can regenerate the CP nucleus to your heart's content while the system is in use and it will have no effect until the real CP is re-IPL'ed. Under NO circumstances should you EVER IPL that CP system disk in a virtual machine that has write access to the disk, as that would guarantee complete and immediate disaster. The result would be that two copies (one real and one virtual) of CP would be paging and SPOOLing and recording errors on the same disk areas and at the same time. It is quite permissable, however, to define separate areas on dedicated mini-disks, and generate a VM-under-VM for test purposes. This is, in fact, an excellent way to test out a new or changed version of CP, before commiting it to production. You would do well, however, to run that virtual VM/370 in a virtual machine with class G privileges only, as you run the risk of sending critical operational commands (such as SHUTDOWN) to the first level (real) CP which could play havoc with your user community. So long as you are a class G user, such privileged commands will be rejected by first-level CP, but will be quite acceptable to your second-level (virtual) CP.

## On CMS

What was said about a CP nucleus generation having no immediate effect upon ongoing system operation is NOT true of CMS. Whenever you are making changes to the CMS system disk you should either be alone in the system. The reason is, that when a user IPL's CMS, a file-directory to the CMS system disk is either set up in the user's virtual storage, or - more probably - if you use the standard DCSS's (Discontiguous Shared Segments), the S-Disk directory is made available to the user in a shared manner. Whichever is the case, when you alter files on the S-disk, those alterations are not known to the other CMS users on the system, and at some point a user will enter a command or invoke some CMS function which requires an S-Disk access, and - poof! - things are not where that user's in-storage directory says they are.

Also remember that if you make use of those DCSS's, that any change to the S-Disk requires that you re-install the DCSS's. And whenever you re-install a Named System or a DCSS you should make very sure that no other user is currently using those segments. Otherwise, your users run the risk of winding up with mixed segments - some old pages, some new - if there are any pages not yet referenced in the in-use segment.

Probably a few words are in order concerning NAMED SYSTEMS and DCSS's (Discontiguous Shared Segments).

The VM/370 system, as a performance option, allows you to share certain read-only segments of storage which may be common to many users. In fact, you can share segments in an unprotected mode (read-write) but that goes beyond the scope of this discussion.

NAMED SYSTEMS and DCSS's are conceptually similar, excepting that NAMED SYSTEMS can be IPL'ed by name, where as DCSS are not IPL'able systems, but segments of storage which exist above the user's normally addressable storage and may be "attached" for use by many users concurrently. The areas defined in DMKSNT which are used to contain these segments are in reality "permanent paging" areas, in that segments are installed by loading them into virtual storage and then - via appropriate CP commands - "paging" them out to the predefined System Name Table (DMKSNT) disk areas.

To INSTALL DCSS's, you MUST have addressability to the storage areas at which the DCSS's reside. To USE DCSS's, you MUST NOT, have addressability to the storage areas occupied by the DCSS's. A user whose virtual storage size overlaps with a DCSS, will not be allowed to "attach" that DCSS. Therefore, if you use the standard system location for CMSZER, any user with a virtual storage greater than 960K will not be able to take advantage of that DCSS.

There are three primary CMS shared segments. One is the named system CMS which most users will IPL rather than issuing an IPL to 190. Try the two different approaches some time one after another and notice how much faster IPL CMS responds (it is a page-in operation), than does IPL 190 (which is a virtual IPL with all of the overhead of clearing and resetting and bootstraping).

The other two primary CMS segments, CMSSEG and CMSZER, are "attachable" DCSS's which contain high-use CMS modules such as editors and OS simulation routines in CMSSEG, and segment zero sharable code as well as (optionally) S-Disk and Y-Disk directories in CMSZER.

Two system EXEC's are provided to load the required modules and issue the SAVESYS for CMSSEG and CMSZER. CMSXGEN EXEC loads and saves CMSSEG, and CMSZGEN EXEC loads and saves CMSZER. You need only invoke these EXEC's and specify the load address of the segment to be saved (if you use the recommended DMKSNT entries for those segments, the hexadecimal addresses are F0000 for CMSZGEN, and 100000 for CMSXGEN). You will be prompted by CMSZGEN as to whether you wish to save the S-Disk and Y-Disk directories. You should reply yes to the S-Disk question, but be careful about the Y-Disk. If you are using IPF (the Interactive Productivity Facility), and have all of its panels on the Y-Disk, the directory for the Y-Disk will probably not fit in the system-recommended CMSZER area and you should reply "NO".

The re-installation of the NAMED SYSTEM CMS is done simply by IPL'ing 190 and replying SAVESYS CMS when CMS issues its VM READ following IPL. CMS will save itself and then IPL the NAMED SYSTEM.

Keep in mind that the disk areas which will contain these NAMED SYSTEMs and DCSS's are "hard-wired" in the CP module DMKSNT as to actual volume, cylinder, and page location. The virtual machine that you are using to install these segments must therefore have write access to that volume (or volumes), and at the virtual address(s) specified in DMKSNT ASSEMBLE.

The whole area of VM/370 source maintenance and the application of temporary fixes or local modifications is really much more simple than it may at first appear.

What is required is an understanding of the CMS UPDATE command and its functions, and a close look at the VMFASM EXEC, which simply wraps some standards around the usage of the UPDATE command.

## The UPDATE Command

The CMS UPDATE command is a powerful utility program designed to apply modifications to source program files. It has many options which may be selected depending upon the requirements of the user. It can function rather simply as a "controlled merge" of one file into another (single level update) where an original source file is updated by a second file containing control records and/or new source statements. Or it can function as a very powerful utility, selectively drawing together control and update information from a great many sources to result in a complex update (multi-level update with auxiliary control files).

For a full understanding of the UPDATE command one should read the appropriate manuals. For our purposes here, we will concern ourselves with the somewhat formalized use of the command within the VMFASM EXEC and the application of source maintenance to CP and CMS.

When the UPDATE command is invoked, two file-ids are passed to the command. The first is the id of the file to be updated (the original source file) and the second is the id of either a simple update file if the NOCTL default option is selected, or - as in our case - the name of a control file if the CTL option is selected.

The default filetype for a control file is CNTRL, and the file consists of two basic types of records. The first record, which is required, is the MACS record. This is an information-only record, and is only used by UPDATE if the STK and CTL options are selected - as they are in our usage via the VMFASM EXEC. The MACS record has the following format -

```
ULI MACS m1 m2 ... m8
```

```
where - ULI is the Update Level Identifier (be patient).
        MACS identifies this as a MACS record.
        m1 to m8 are the filenames of up to 8 MACLIB's.
```

Thus, the following MACS record from the DMKSP CNTRL file -

```
TEXT MACS DMKSP DMKMAC DMSSP CMSLIB OSMACRO
```

identifies a MACS record with an Update Level Identifier of "TEXT", and five macro library (MACLIB) names specified.

Records which follow the MACS record are called "file records" and, depending upon content, can request various actions. File records have the following format -

```
ULI UAID PA1 ... PAn
```

where - ULI is the Update Level Identifier (again, be patient).
UAID is the Update or Auxfile Identifier.
PA1 thru pan are optional Preferred Auxfile Id's.

Thus, the following file record from the DMKIPF CNTRL file -

```
TEXT AUXR60 AUXB20
```

is a file record with an Update Level Identifier of "TEXT", specifying an Auxfile of AUXR60, and one Preferred Auxfile Identifier of AUXB20.

It should be noted that the Update Level Identifier "PTF" is treated as a special case and should not be specified without an understanding of its usage.

Now, with all of the pieces of a Control File identified, let's take an example, examine it, and see just how it is used. I have chosen the DMKIPF CNTRL file used by the Systems IPO/E packaging of VM/370, as it is a more complex control file than the ones used for non-Systems IPO/E systems, and is of a format which illustrates the construction of a control file for applying local (customer) modifications. The DMKIPF CNTRL file is as follows -

```
LCL MACS CPBSE DMKMAC CMSBSE CMSLIB OSMACRO
LCL AUXLCL
TMP AUXTMP
IPF AUXIPF
TEXT AUXB20
TEXT AUXR60 AUXB20
```

Now let us assume that the following command string is entered at the terminal -

```
UPDATE DMKXYZ ASSEMBLE A DMKIPF CNTRL A (CTL
```

This will invoke the UPDATE utility and specify that the assemble source program DMKXYZ on the A-Disk is to be updated according to directions contained in the control file (since we have specified option "CTL") DMKIPF CNTRL, also on the A-Disk.

The source program DMKXYZ will be loaded into storage prior to performing the updates. There is a STOR/NOSTOR option for the UPDATE command and STOR is the default if CTL is specified. The assumption here is that if a simple update is to be performed, it is only a merge of one file into another and performing the operation on disk is as efficient as doing it in storage. When CTL is specified, however, it is likely that many updates are to be forthcoming, which would require re-writing the source file to disk many times. Thus, if the source program will fit in the virtual storage available, it is read in once, all updates applied, and then written to disk. If the program is too large to fit in the virtual storage available, NOSTOR will be used, and the update may take considerably longer.

UPDATE will now locate the DMKIPF CNTRL file and work its way from the last record in that file to the first. In our example the last record is the file record "TEXT AUXR60 AUXB20" which, according to our formats, specifies one Preferred Auxfile "AUXB20". UPDATE will now search for a file named DMKXYZ AUXB20 - the source program name with a filetype as specified on this file record. If that file is located, NOTHING is done and this file record is ignored. If it is not located, then the search continues for a file named DMKXYZ AUXR60. This is the function of a Preferred Auxfile. In effect it says, "if any of the Preferred Auxfiles specified on this file record are present, ignore the file record, otherwise treat it as an ordinary file record and continue processing with the auxfile specified" - AUXR60 in our example.

This may seem a bit strange, but now look at the next to last file record "TEXT AUXB20". The Preferred Auxfile procedure allows us to selectively apply updates depending upon what files are available. In our example, DMKXYZ AUXR60 will only be applied if DMKXYZ AUXB20 is not available. In no case will they both be applied.

And so the process continues from bottom to top, searching in turn for files named DMKXYZ AUXIPF, DMKXYZ AUXTMP, and DMKXYZ AUXLCL. Note that in all cases the filename applied is the same as the filename of the source program to be updated.

The key to a multi-level update with auxiliary files is in the three characters "AUX" in the identifier specified on the file record. If the first three characters are anything else, the UPDATE utility will assume that the filetype pointed to is an update file, not an auxiliary file.

An auxiliary file is not in itself an update file, but is a pointer to the actual update files. The records in an auxiliary file are quite simple - there is one record for each update file to be applied and that record specifies the FILETYPE of the file which contains the updates. The FILENAME, as before, is assumed to be the same as the filename of the source program being updated. And, as before, the update files specified in an auxiliary file are applied to the source from the bottom up.

Let's now sharpen the focus by looking at all of the pieces together. Assume that the following files are accessable to the UPDATE program.

```
DMKXYZ AUXB20
   (which contains the following records)
S12345DK 999 FIX TO BROKEN WIDGET       11/21/81
S12335DK 110 FIX TO BENT THUNDERBLAST   10/19/79
   (note that anything past the first blank is comments)


DMKXYZ AUXR60
   (which contains the following record)
S11345DK 999 FIX TO CRACKED BLIVOT      10/04/78


DMKXYZ AUXLCL
   (which contains the following record)
L99999DK 000 LOCAL CHANGE TO BENT WIDGET 11/24/81 RWB


DMKXYZ S12345DK
   (which contains the actual update data and control records)


DMKXYZ S12335DK
   (which contains more actual update data and control records)


DMKXYZ L99999DK
   (which contains even more update data and control records)


DMKXYZ S11345DK
   (which contains ... you get the picture.)
```

And now to continue the UPDATE process that we began above, the following takes place. Since the first (last actual record - remember, bottom to top) record in our DMKXYZ CNTRL file specified a preferred auxiliary file of AUXB20 and, since a file named DMKXYZ AUXB20 exists, UPDATE will not even look for DMKXYZ AUXR60 even though it exists. Therefore the file DMKXYZ AUXR60 and the file to which it points, DMKXYZ S11345DK, will be ignored.

The next file searched for will be DMKXYZ AUXB20 (second to last in our control file) and when this file is found, it in turn points to update files DMKXYZ S12335DK and DMKXYZ S12345DK which will be applied in that order (remember, bottoms up).

The control file search then continues for files named DMKXYZ AUXIPF and DMKXYZ AUXTMP, neither of which exist in our example, but which may exist with different filenames for application to other source modules. Our last "hit" will be the file DMKXYZ AUXLCL, which in turn points to file DMKXYZ L99999DK which will be applied against our source program.

At this point, the resulting updated source program will be written to disk but, since we have taken the UPDATE option default "NOREP", the name of the updated version will be changed to a $ followed by the first seven characters of our filename. Thus, our resulting updated source program is a file named $DMKXYZ ASSEMBLE.

You have been very patient to this point, so let's now discuss the MACS record and those Update Level Identifiers (ULI). In fact, in our example they have absolutely no use, as we accessed the UPDATE command directly rather than having it imbedded within an EXEC as is the case with VMFASM and VMFMAC.

Had we accessed the command with the option "STK", upon completion of the update, the UPDATE utility would have placed the following line in the CMS console stack -

```
* LCL CPBSE DMKMAC CMSBSE CMSLIB OSMACRO
```

The asterisk (*) is placed first in the line so that if the STK option is selected from a terminal rather than an EXEC, the stacked line will be treated as a comment and not generate an error. The next token "LCL" is the update level identifier on the last file record for which a "hit" was found. In our case it was the line "LCL AUXLCL" since we had a file called DMKXYZ AUXLCL. If no "hits" had been found, the update level identifier of the MACS record would have been used (which in our example would have been the same - LCL). The next five tokens in the stacked line of our example are the MACLIBs specified on the MACS record.

We shall see just how this information is used by the VMFASM and VMFMAC EXEC's later in our discussion.

We will now take a brief look at the Update Control Statements. This is not intended to be an in-depth tutorial on the use of the utility and, for particulars on the control statements, the "CMS Command and Macro Reference" manual should be consulted.

All Update Control Statements are identified to the UPDATE utility by the characters ./ which must be present in columns 1 and 2. There are five control statements - S, I, D, R, and * - to specify re-sequencing, inserting, deleting, replacing, and comments respectively. The I and R control statements have an optional $ operand which is worth noting as this requests that UPDATE resequence the inserted or replaced lines. If this operand is not used, sequence errors may result in the updated source program which can have serious (and sometimes undetected) effects where multiple updates are involved.

The UPDATE utility, as you can see, is not really difficult to use, but it does offer considerable flexibility which may be reason for some confusion. We have not covered the full UPDATE function, nor discussed all of the options, but we have covered enough to see that a simple change in control files, or in the way in which disks are accessed, can produce totally different versions of a source program.

## The VMFASM EXEC

VMFASM is an EXEC which is supplied primarily to perform VM/370 maintenance functions. Once its usage is understood, however, it is quite useful either as is, or in some modified version as a tool for many program development and maintenance applications.

VMFASM does some initial housekeeping by setting up defaults for the assembler and checking to determine that both the source program and the control file, which were specified when it was invoked, exist.

VMFASM then invokes UPDATE, and three non-default options are selected by the EXEC: PRINT, which produces an update log on the virtual printer; CTL, which informs the UPDATE command that this update is to consist of a multi-level update (the update file specified is a control file and not a simple update file to be merged with the source file); STK, which requests that the UPDATE command stack certain information concerning the outcome of the update in the CMS console stack.

Upon return from the update utility, condition codes are tested and if no updates existed for the program to be assembled, the actual program name is passed to the assembler. If updates were applied, the $program-name as created by the update utility is the updated program to be assembled.

That line that we looked at earlier, which is placed in the CMS console stack by UPDATE, is then read (&READ ARGS) and a little MODULE named VMFDATE is executed once for each MACLIB filename passed back in the console stack. The VMFDATE module will place in a disk file, name, location, and date and time of last change information for any file specified. It is used here to capture documentary information which will later be prefixed to the resulting TEXT deck along with information from the AUX-FILES used during update. This provides an excellent audit trail right within the TEXT program of all updates applied and the currency of the macro libraries used in the assemble.

The program is now assembled and, if the assemble is successful, the resulting TEXT deck is prefixed with the audit information and renamed according to the Update Level Identifier that was returned in the console stack from UPDATE. If this ULI is anything other than TEXT, it is prefixed with the characters TXT and, following our example above, if we were to enter the following -

```
VMFASM DMKXYZ DMKIPF
```

the result would be an object program named DMKXYZ TXTLCL.

There is one other caveat in the use of VMFASM. If the source program to be assembled is on any read/write disk other than the A-Disk, the resulting TEXT program will not be renamed according to the Update Level Identifier, nor will the audit information be added to the object program.

## The VMFMAC EXEC

The VMFMAC EXEC is not nearly so commonly used as the VMFASM EXEC, and you may never have occasion to use it at all. It is an EXEC of which you should be aware, however, and may also prove useful as a prototype for developing maintenance EXEC's for other uses.

If you look at any VM/370-supplied macro library (MACLIB), you will find that there is also an EXEC file supplied with the same name as the MACLIB. There is, for example, a DMKMAC MACLIB and a DMKMAC EXEC and, corresponding to CMSLIB MACLIB, you will find a CMSLIB EXEC. If you look at these EXEC's you will find that they provide lists of the MACRO and COPY files which make up their corresponding MACLIB, and that they are in the format of a CMS EXEC file as produced by the LISTFILE command with the EXEC option.

The VMFMAC EXEC uses these EXEC's and a control file to apply updates to the individual MACRO and COPY files and then completely recreates the MACLIB. The function of the EXEC is much the same as the function of VMFASM, except for the important difference that VMFMAC updates ALL of the macros and copies in a given MACLIB, and therefore all of the MACRO and COPY files which make up a MACLIB must be available and accessed when the EXEC is invoked.

## A Scenario

Perhaps it would be helpful to describe a typical maintenance process. Let us suppose that you discover a problem with your system, and through careful research you decide that it is somehow related to a dented widget. You place a call to the IBM Support Center and explain that you have a new problem which is CP-related. You are assigned problem number 0X123 for tracking purposes, and connected to a level one support person.

You explain the problem, and a search of the data-base reveals that there is indeed a known fix for your widget problem. It is PTF number 12346, and it is a change to CP module DMKXYZ. It is also quite short, so rather than mailing the change, the level one person reads the fix to you on the phone.

You are sitting at your terminal and you can key in the fix directly. You already have the information you will need to create the update file with whatever editor you may use, so you prepare for the fix by editing a new file named DMKXYZ D12346DK. (Those last two characters of the filetype would be DS, rather than DK if the fix were for a DMS (CMS) module - convention is nice). You then key in the update information as it is read to you - perhaps -

```
 ./ * FIX FOR DENTED WIDGIT
 ./ R 2000 2000 $ 2100
 XNAME   LM   12,14,XADDRES  LOAD REGS 12 THRU 14.
```

You thank level one for the fix, and promise to call back for more info if it doesn't fix the problem, or to close the incident if it does.

(If you have the Program Product "Information/System for VM/370" at your installation, you would probably have researched the problem on your system and found the fix without the call to IBM.)

You still have a bit of research to do, as you have a local modification to that DMKXYZ module, and you must assure that it does not conflict with the IBM fix. You perhaps don't have the VM/370 source online, so you attach a tape drive to your MAINT virtual machine, mount the CP Source Tape which is in your tape library, and issue the command -

```
VMFPLC2 LOAD DMKXYZ ASSEMBLE
```

This will load the assemble source to your A-Disk. You now access the 194 disk as your B-Disk to get access to the necessary MACLIB's, and you access the 294 disk as your C-Disk to get access to those other AUX and update files that we looked at earlier.

You will now edit that file DMKXYZ AUXB20 that we looked at earlier and that currently contains the following records.

```
S12345DK 999 FIX TO BROKEN WIDGET      11/21/81
S12335DK 110 FIX TO BENT THUNDERBLAST   10/19/79
```

You will add to the TOP of that file the following record -

```
D12346DK 999 FIX TO DENTED WIDGIT 01/01/82 FROM IBM SC. RWB.
```

So that file DMKXYZ AUXB20 now looks like this -

```
D12346DK 999 FIX TO DENTED WIDGIT 01/01/82 FROM IBM SC. RWB.
S12345DK 999 FIX TO BROKEN WIDGET      11/21/81
S12335DK 110 FIX TO BENT THUNDERBLAST  10/19/79
```

You are now ready to assemble the module with the fix, so you enter the following -

```
VMFASM DMKXYZ DMKIPF
```

Your new DMKXYZ TXTLCL object deck is created which you will probably copy to the 194 disk after renaming the old copy to - perhaps - DMKXYZ OLDTXT just in case the fix doesn't work and you wish to easily back it out.

Now all that is left is to regenerate the CP nucleus as described earlier, re-IPL the system at some convenient time, and test the fix. You find it works just fine, so you call back the IBM Support Center, this time giving incident number 0X123, thank them kindly, and ask them to close the incident.

The filetype for a VM/370 update file is customarily made up by an S prefix, followed by the PTF number, and a suffix DK for CP (DMK) modules or DS for CMS (DMS) modules when the fixes are supplied via PUT or PLC tapes. We have changed the prefix S to D to identify an IBM fix not included in a PUT, and to L to identify local (non-IBM) modifications. This allows for easy indentification. Remember, it is your responsibility to determine whether or not any non-PUT/PLC changes (either IBM-supplied or local) are still needed when you apply a new PUT or PLC tape. This is a good reason for having your own version of the CNTRL file to produce TXTLCL decks rather than just TEXT. The TEXT decks will most likely be replaced with the application of a PUT/PLC tape while TXTLCL decks will not. Still, you MUST research to determine if a fix still "fits", and then re-assemble all locally altered modules to assure that any changed macros or control blocks are properly updated.

THIS PAGE INTENTIONALLY LEFT BLANK

I will now outline my own personal procedures for generating an unmodified CP and CMS nucleus upon the arrival of a new PUT tape. It is not the only way, just my way.

I will logon to the MAINT machine and attach a tape drive as virtual address 181 and mount the PUT tape. Then I issue a VMFPLC2 LOAD to get the VMSERV EXEC and the various memos loaded to MAINT's 191 A-Disk. Then I access the 191 disk as the C-Disk which the VMSERV EXEC prefers to run from (because VMSERV is going to access many disks as the A-Disk as it proceeds to load the various updated files and if it were running from the A-Disk, it would cut its own head off when it accessed another disk as the A-Disk).

The first execution of VMSERV loads in various documentation and discovers what products are represented on the PUT tape. I normally respond "NO" to the question about whether I want memos printed, as I can read them quite nicely at the terminal. It is politic at this point to examine the PUT memos to see if there are any changes from previous ways of doing business.

VMSERV is then invoked a second time, and I respond to the prompting questions as to what things and where I want loaded. In our section concerning MAINT's disks, we pretty well covered what goes where, and if you prefer (or, as has happened to me on occasion, you find you have a bad file on the tape which prevents VMSERV and the product EXEC's which VMSERV invokes from operating successfully) you might print off a copy of the product documentation which shows the contents of each file on the tape, and simply access the particular disk to be loaded as the A-Disk (194 for CP TEXT, 293 for CMS AUX, etc.) and load them yourself using VMFPLC2.

You might also choose to selectively load the CP files to the 194 and 294 disks and leave the CMS updates for a later time, remembering that modifications to the 190 disk will affect other system users.

Once I am satisfied that all of the updated files have been loaded to the appropriate mini-disks, I re-assemble the installation-tailored modules (DMKSYS, DMKRIO, DMKSNT, etc.). This is generally not necessary, but on occasion, changes are made to system macro libraries, and considering how little time it takes to do three or four assembles, I consider it to be faster to do them than to research the necessity.

I am now ready to re-generate my CP nucleus, which the following sequence of commands accomplishes -

```
        ACCESS 191 A              - Just for fun.
        ACCESS 194 B              - To get access to the TEXT decks.
```

```
    PURGE READER             - Make sure the reader is empty.
    SPOOL PUNCH *            - Get that nucleus sent to me.
    SPOOL PRINTER *          - Get the load map sent to me.
    VMFLOAD CPLOAD DMKSP     - Punch out that nucleus.
    IPL 00C                  - IPL the new nucleus.
    IPL 190                  - Re-IPL CMS.
    ACCESS 194 B             - Get 194 for my load map.
    READ CPNUC MAP B         - Save load map for debugging.
```

And that takes care of CP - all done.

Now for CMS (I wait until everyone's left). Then make sure all the updated TEXT, MACLIB, MODULE, HELP Screen, and EXEC files have been loaded to my 190 disk. And then -

```
  ACCESS 191 A               - Just for fun.
  ACCESS 190 B/A             - Get read access to ALL the TEXT
                               files.
  PURGE READER               - Make sure the reader is empty.
  SPOOL PUNCH *              - Get the nucleus sent to me.
  SPOOL PRINTER *            - Get the load map sent to me.
  VMFLOAD CMSLOAD DMSSP      - Punch out that nucleus.
  IPL 00C                    - IPL the new CMS nucleus.
  Answer a lot of questions from CMS.
  (Remember to have that relative address available)
  IPL 190 PARM SEG=NULL      - Re-IPL CMS without DCSS's.
  ACCESS 190 A               - Get read/write access to 190.
  CMSGEND ASSEMBLE           - Re-generate the Assembler Aux.
                               Directory.
  Answer another question or two (or take defaults).
  IPL 190 PARM SEG=NULL      - Re-IPL CMS without DCSS's.
  ACCESS 190 B/A             - Get read access to ALL 190 files.
  CMSXGEN 100000             - Re-install CMSSEG.
  CMSZGEN F0000              - Re-install CMSZER.
  Answer a couple more questions.
  IPL 190                    - Re-IPL CMS.
  SAVESYS CMS                - Re-install the NAMED SYSTEM CMS.
  ACCESS 194 B               - Get a place to save the load map.
  READ CMSNUC MAP B          - Save load map for debugging.
```

And basically, that's it. Of course I may need to re-install a number of other DCSS's for DOS, or VSAM, but the above steps (provided I have loaded all the right stuff on my 190 and 194 disks) take care of my basic CP and CMS maintenance.

Once the new files are loaded, the procedures listed above take perhaps 20 to 30 minutes - not so bad. Then, a silent prayer, SHUTDOWN and re-IPL to test success.

I keep all my ASSEMBLE files (DMKRIO, DMKSYS, DMKSNT, DMKBOX, and any source that I have had to modify because of local fixes) on MAINT's 191 disk. After I have assembled any source, I will copy the resulting TEXT (object) deck either to my 194 disk (CP) or to my 190 disk (CMS), and erase it from my 191 disk. If there is any doubt as to the eventual success of a modification, I will RENAME the old TEXT module to something like TXTOLD before copying the new TEXT to the 190 or 194 disk. That way, I can always back out a change without re-assembly.

I feel that by having only one copy of a TEXT deck available gives me peace of mind in that I cannot inadvertantly (by having disks ACCESSed in the wrong order) include the wrong copy of a module in a CP or CMS nucleus.

THIS PAGE INTENTIONALLY LEFT BLANK

# POSTLUDE

What has herein been described with many words, can really be reduced to saying that both CP and CMS are - in the main - made up of a few hundred program modules which we combine - and customize via three or four assemblies - to create a system that is acceptable to our needs.

What has been said holds true not only to the application of system updates, but to the generation of new systems as well, be they unmodified VM/370 SCP's, BSEPP, SEPP, VM/SP, and (let us hope) new things to come.

Get your disk allocations in order, combine the right bunch of TEXT decks in the right order, and you have conquered VM/370 maintenance.

THIS PAGE INTENTIONALLY LEFT BLANK

READER'S COMMENT FORM

|  |  |
|---|---|
| Title: | VM/370 MAINTENANCE MADE SIMPLE<br>Washington Systems Center<br>Technical Bulletin GG22-9277-00 |

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Please state your occupation: ______________________________

Comments:

Please mail to: G. E. Hollendursky
IBM Washington Systems Center
18100 Frederick Pike
Gaithersburg, MD 20879