**10. MAINFRAME MODULES**

**10.1 Introduction to Mainframe Modules**

**10.2 Source Modules**

**10.3 Object Modules**

**10.4 Program (Load) Modules**

**10.1 Introduction to Mainframe Modules**

The computer revolution has brought about many changes in the way information is processed. The whole idea of data processing is to get a machine to perform most of the tedious, repetitive functions involved in the processing of information. The computer is efficient, reliable, fast, and accurate; it's actually less costly than utilizing human resources in the long run since it saves on labor and waste.

The very first computers were programmed in machine language (ones and zeroes), which was very difficult and time-consuming for the programmers. They had to take into consideration the actual main memory storage locations within the machine in which the program would run. Imagine, if you will, coding an Assembler program without using labels or base registers, dealing, instead, exclusively with absolute addresses.

Eventually, someone realized that since the computer itself was so good at performing repetitive tasks, it should be able to alleviate some of the tediousness involved with programming in machine language. This was the start of the development of the assembly program, which would allow abbreviations to be used instead of numbers (like 'L' for the LOAD instruction instead of the op code '58'). After this was the development of the compiler, or computer language translator, which uses English-like phrases instead of abbreviations.

The early computers only allowed for one program to execute at a time. This meant that all system resources were reserved for this program alone, whether it needed to use the resource or not. This idle waste of resources impeded production. With the development of multi-programming techniques, two or more programs would be allowed to execute concurrently (not simultaneously) and all programs would share system resources. Programs running concurrently take turns at executing. For example, the first program in a group of two would begin execution, until an I-O access was requested by this program. The program's execution would have to be placed in a "wait state" by the operating system until the I-O request is completed. At this point, instead of allowing the CPU to remain idle, the second program in the group would begin executing until it had an I-O request.

When the second program is placed in a "wait state" by the operating system for an I-O request, the first program would be allowed to continue its execution providing its I-O request had been completed. This process continues until both the programs have finished. Taking turns at executing is the premise behind the technique of multiprogramming. The operating system at Marist (z/OS), is a multiprogramming system.

Before a program can execute, it must be loaded into the computer's main memory. The program remains in memory until it has finished executing. With multiprogramming, several programs are in memory at one time while each one takes its turn at executing. Each program loaded into memory has a fixed starting location, which means that once execution begins the program resides at the same address locations (it is not moved) until it finishes.

The operating system divides the addresses of main memory into regions. A load module should be able to run in whichever region the operating system places it, regardless of the address. To accomplish this, all addresses within a load module must be expressed as relative to any given starting address. When an object module is created, by either a compiler or an assembler, displacement addresses (relative addresses) are generated. Consider a program which has a data item to be saved into a memory location. The memory location is 100 bytes beyond the starting point of the program. So if the program were loaded into memory at address 001000, the data item would be located at address 001100 (001000 + 100). If the program were loaded into memory at address 002400, the data item would be located at address 002500 (002400 + 100). The program should refer to the data item at the relative address of 000100, rather than the absolute address of 001100 or 002500. The program is then relocatable, capable of being loaded into memory at any given absolute address.

The reason for dividing the computer's memory into regions is so that more than one program may be in memory at any one given moment, each program residing within a region of its own. Each program in memory takes its turn at executing. Consider a computer system with four regions of memory. Five programs are waiting to execute. Four programs are loaded into the four regions of memory, each one taking a turn at executing. The fifth program will be loaded into whichever one of the four regions becomes available first, depending on which of the other four programs finishes first. Since each region of memory has a unique range of memory addresses, and since the operating system will dynamically load a program into the first available region, a program must be relocatable.

Remember, a program loaded into the memory of a computer must be in the form of a load module. The load module contains the actual machine code generated by a compiler or an assembler, plus the addresses of additional external modules needed to execute the program load module.

IBM supplies a program, known as the LOADER, that will produce a load module from an object module, load it into any valid region in main memory, and execute it.

**10.2 Source Modules**

Coming soon

**10.3 Object Modules**

The LOADER program accepts as input the program source code which has first been translated into relocatable object code; in other words, the LOADER accepts the object module as input. The LOADER then link edits the object module and any other requested object modules into a load module. This means that the original object module and the addresses of any load module(s) that it references are grouped together into one module with all addresses resolved. The load module is placed into a region of memory where the loader relinquishes control to the program so that it may execute.

Once execution has been completed, the object module will no longer exist, nor will the load module. The LOADER does not save the object module or the load module. If the program needed to be executed again, it would need to be resubmitted to the LOADER which would then start the entire process over again. As you can probably imagine, this process is too time-consuming to follow through each time a program needs to be executed. The LOADER's main function is to provide a convenient means of testing a new or altered program. Why take the time to re-compile (or re-assemble) and re-resolve addresses every time a program needs to be executed if there have been no changes made to the program? This is where the BINDER (LINKAGE EDITOR) comes in.

**10.3 Program (Load) Modules**

Let us assume that you are a programmer in a fairly large shop currently working on a program that is to produce a report for upper-level management. Among other things, the report is to include the date for which it was produced. Given to you through a parameter on the EXEC card is the report date in Julian format (i.e., in the form YYDDD where YY are the last two digits of the year and DDD represents the DDDth day of that year). Management probably does not understand what a Julian date is, or even if they did, they expect to see a date like "MONDAY, AUGUST 23, 2010" in the heading of the report. Given the Julian date, you must somehow convert it to the English-type date illustrated. Doing so is a long and complicated process, but you must do so if the report is to be acceptable. You must sit down and write the code...or must you?

As it turns out, a programmer in your shop had the same problem a few years back. He wrote a subroutine with well-defined, input and output parameters that would perform the necessary date conversion. To save yourself some trouble, you could copy in his source code during assembly or compilation (the COBOL 'COPY' statement), and use his subroutine in your program (with using the COBOL 'PERFORM' verb). However, this is inefficient. Since the subroutine has already been written and compiled, it would be more efficient to access the object module of the subroutine instead of the source code.

IBM supplies a program which supports accessing a previously compiled program or subroutine in object module form. It is the BINDER (LINKAGE EDITOR). It has the ability to take several separate object modules and link them together into one load module which can then be executed.

The BINDER (LINKAGE EDITOR) program accepts an object module as input. The input object module plus any other object or load modules referenced by it (through the COBOL 'CALL' verb) are linked together to form one executable module, the load module. The referenced object modules are actually grouped together with the input object module to form the load module, whereas only the addresses of the load modules referenced are resolved in the load module, provided a SYSLIB DD statement is included in the JCL specifying the location of the referenced load modules. For example, the report program discussed above would be processed as follows: the BINDER (LINKAGE EDITOR) would link the input object module for the report program with the object module for the date conversion subroutine and form one load module. The programmer must inform the BINDER (LINKAGE EDITOR) where to look for any externally referenced modules via a SYSLIB DD statement in the BINDER (LINKAGE EDITOR) job step.

The BINDER (LINKAGE EDITOR) forms a load module which it saves into a partitioned data set but does NOT execute it. The load module saved by the BINDER (LINKAGE EDITOR) is ready to be executed at any given moment. It remains permanently available on disk.

Once the load module is stowed, all that it takes to execute it is the invocation of a system program known as Program Fetch. Program Fetch, when invoked, simply goes out to disk, grabs the specified load module, loads it into memory, and then transfers control to it. All that is needed to initiate Program Fetch is this statement:

// EXEC PGM=program name

If your program were named PAYROLL, for instance, the statement

// EXEC PGM=PAYROLL

would invoke Program Fetch, which would then load and execute the program (given that all appropriate DD statements are included).

The differences between the BINDER (LINKAGE EDITOR) and Program Fetch are as follows:

| BINDER (LINKAGE EDITOR) | Program Fetch |

------------------------|-------------------------|---------------|

object module as input? | yes | no |

------------------------|-------------------------|---------------|

brings in external | | |

references? | yes | no |

------------------------|-------------------------|---------------|

saves a load module | | |

onto disk? | yes | no |

------------------------|-------------------------|---------------|

places the load module | | |

into memory? | no | yes |

------------------------|-------------------------|---------------|

allows program to | | |

execute? | no | yes |

-------------------------------------------------------------------

The BINDER (LINKAGE EDITOR) is capable of many more functions than those mentioned here. For more details, refer to IBM's manuals regarding the linkage editor and the loader in the z/OS environment.