The CDC SCOPE program loader, later known as the CYBER Loader, was a complex program the combined the functions of a linkage editor and an image loader. (Linkage editing involves combining relocatable object modules to produce executable programs, as do UNIX's `ld` and Windows' `LINK.EXE`. Image loading involves loading a program into memory, as does UNIX's `exec()` function.)

The CYBER Loader became increasingly sophisticated over the years as a result of CDC's attempts to compensate for the shortcomings of its hardware, namely:

- Limited memory capacity. 18 bits of address space is only 256K words, about 2MB in modern terms. And in the early days, many 6000-series machines did not have even this much memory, due to its high cost.
- Lack of virtual memory. An executing job had to be entirely in memory. That is, there was no hardware support for maintaining a “working set” in memory, with the rest of the program paged out to secondary storage.
- Simplistic memory management hardware. Each program occupied a single, non-shared, contiguous block of memory. You could not map part of a process's address space to a shared library or to system memory. This means that each program needed to have its own copies of any library routines it needed.

To address these limitations, the CYBER Loader provided programmers a variety of approaches to loading. Some were quite complex and required significant effort during program development. Here, we will describe these options in increasing order of complexity (roughly in chronological order).

### Relocatable Load-and-Go

The simplest way to run a program was to tell the system to load a file containing relocatable object modules, link them in memory, and start the program, all in one command. Understanding how this worked requires a little background on how most SCOPE/Hustler compilers worked.

A compiler read a source code file containing one or more program units. (A program unit is a main program, subroutine, or function, with subroutines and functions being nearly the same thing.) The compiler would produce an output file which contained one section of relocatable object code for each program unit. This is similar to the way that compilers in other operating systems work, except that the “sections” were delimited by file marks that were implemented as part of the operating system file structure. For each main program, the compiler produced a special loader table entry which specified a “transfer address”, the offset from the beginning of the routine where execution should begin.

By default, most compilers wrote their output to a file named LGO, which stood for Load and Go. Specifying the name of a local file as a command caused the system to load and execute the contents of that file. Thus, a simple compile-and-run job would look like this:

```
FTN,I=COMPILE. (Read source code from COMPILE, write relocatable to LGO.)
LGO. (Load file LGO and run it.)
```
This capability is rarely seen in modern operating systems. If Windows could do this, for example, then you could directly execute .OBJ files without first linking them into .EXE files.

**Single-level overlays**

Relocatable load-and-go was fine for programs which were usually recompiled before being run. However, once development had settled down and a stable version of a program had been completed, it would have been wasteful to require each user of a program to relink the program from relocatable object each time it was run. For relatively simple stable production programs, it was possible to create a pre-linked version of a program called a (0,0) overlay. A (0,0) overlay was a file containing essentially the memory image of a loaded program, similar to an MS-DOS .COM file. Creating a (0,0) overlay file from a relocatable file was quite simple:

```
LOAD,LGO.  (Load the relocatable object.)
NOGO,MYPROG.  (Create the file MYPROG containing the linked modules.)
```

The resulting overlay file could then be executed just by specifying the name of the file as a command. It would run the same as the relocatable, but would load faster.

Programs created in this way were not generally referred to as “overlayed”, as they were a trivial special case of the full overlaying mechanism described below.

**Multi-level overlays**

Using multi-level overlays involved dividing an application into a hierarchy of subprograms (overlays) that loaded at different addresses in a job's field length. Each overlay was essentially a separate program which had to be invoked from a higher-level overlay. Once called, an overlay started execution at its main program's entry point, ran to completion, then returned control to the calling overlay.

Each overlay was identified by a pair of numbers. There could be up to three levels of overlay: a single main overlay, numbered (0,0); primary overlays which were direct children of the main overlay and were numbered (x,0), where x>0; and secondary overlays, which were direct children of primary overlays and had numbers like (x,y), where x was the number of the parent primary overlay and y>0. Thus, an overlayed program would look something like this:

```
<table>
<thead>
<tr>
<th>Main</th>
<th>Primary</th>
<th>Secondary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(1,0)</td>
<td>(1,1)</td>
</tr>
<tr>
<td></td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>(2,0)</td>
<td>(2,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2,2)</td>
</tr>
<tr>
<td></td>
<td>(3,0)</td>
<td></td>
</tr>
</tbody>
</table>
```

https://codex.retro1.org/  Printed on 2023/12/10 12:44
Both primary and secondary overlays could call subroutines in the main overlay, and secondary overlays could call subroutines in the main overlay or the primary overlay that called them.

A programmer declared a program to be an overlay by placing an `OVERLAY (x, y)` statement in the FORTRAN or COMPASS source code just before the group of routines that constituted that overlay. An overlay was invoked by a statement like `CALL OVERLAY(1, 0)`. Note that you could call overlays only in a direction away from the root; therefore, `CALL OVERLAY(2, 0)` from within a (1,0) overlay would be illegal.

The overlay capability dated back to the early days of the operating system, but was rarely used by programmers outside of the systems programming unit, because it required quite a bit of effort.

**Segmented programs**

Segmentation was another approach to dividing an application into separately loadable modules. Like overlays, segmentation divided an application into hierarchical groups of modules which were loaded dynamically. Unlike overlays, segmentation:

- Involved a single program controlling groups of subroutines (segments), not a groups of programs (overlays).
- Allowed calling of subroutines both up and down the hierarchy. For instance, a routine in the root segment could call a subroutine in a child segment.
- Did not require source code modifications.
- Was not limited to three levels of hierarchy.

Segmentation did require the preparation of a complex input file containing “segmentation directives” describing the tree of segments, and describing which routines should be in each segment. A segment could not call a routine that belonged to a different segment at the same level, because the two segments would not be in memory at the same time. However, you could place separate copies of a subroutine in different segments, as long as you were aware that the different copies of the routine would be using different local variables.

Segmentation was introduced later than overlaying. It was also slightly less efficient than a skillfully overlayed program. As a result, few system programs used segmentation. For that matter, few user programs did.

I once wrote a program that examined loader maps and produced segmentation directives that would create an admittedly non-optimal segmentation scheme for the program. I suppose that many programmers at other sites did the same thing.

**Overlay capsules**

Overlay capsules (OV CAPs) were a capability introduced late in the career of the CYBER Loader. They were an attempt to combine the flexibility of segmentation with the performance of overlays. OV CAPs were groups of subroutines, while OVERLAYs were main programs and could pass parameters only through COMMON blocks.
There were three routines that your program called to use overlay capsules: LOVCAP, XOVCAP and UOVCAP (load, execute and unload). It was legal to call XOVCAP without a preceding call to LOVCAP.

Unlike CALL OVERLAY, CALL XOVCAP could pass parameters. For example, to call an OVCAP named ADD with a parameter named ADDED—that is to say, to do the equivalent of the normal FORTRAN

\[
\text{CALL ADD(ADDED)}
\]

-you would use these statements:

\[
\begin{align*}
\text{CALL XOVCAP('ADD',ADDED)} \\
\text{CALL UOVCAP('ADD')} \\
\end{align*}
\]

The parameters were passed as a variable length argument list.

OVCAPS required the use of an initial OVERLAY statement in the source code. The initial OVERLAY statement had the format

\[
\text{OVERLAY(file,0,0,OV=nn)}
\]

where \( nn \) was a count of OVCAPs and was used to reserve space for an index to allow random access to the overlay/OVCAP file.

After the OVERLAY statement, there were OVCAP. statements in the source between the (0,0) overlay and each of the subsequent subroutine groups. For example:

```fortran
OVERLAY(file,0,0,OV=2)
PROGRAM BOB (INPUT=15/137, OUTPUT=25/240)
CALL XOVCAP('FOO',A,B,C)
CALL UOVCAP('FOO')
CALL XOVCAP('BAR',I,M)
CALL UOVCAP('BAR')
END

OVCAP.
SUBROUTINE FOO(A,B,C)
A=B+C
CALL INC(A)
RETURN
END

SUBROUTINE INC(A)
A=A+1.0
RETURN
END

OVCAP.
SUBROUTINE BAR(I,M)
I=I .AND. COMPL(MASK(M))
RETURN
END
```

OVCAPs were prelinked but not relocated. Relocation was done at runtime. For some reason the OVCAP runtime loader couldn't do negative relocation. Usually this wasn't a problem except that some of the system routines used negative relocation and couldn't be put in a OVCAP and so had to
be located in the 0,0 overlay. OVCAPs could load other OVCAPs and weren't limited to two levels like overlays.

(Thanks to Robert Turnbull for the discussion of overlay capsules.)