```
*****************************************************************
*                                                               *
*    The New Dynamic Overlay Scheme........for BDS C v1.4      *
*                       August, 1980                            *
*                                                               *
*****************************************************************
```

In order to allow C programs to be longer than physical memory, without resorting to "exec" or "execl" (which may indeed get the job done, but resemble "chain" operations more than true segmentation tools), a new set of capabilities has been built into the CLINK program. Normally, the run-time environment of an executing C program looks like this:

```
--------------------------------------------------------------------
low memory:  base+100h:  C.CCC run-time utility package (csiz bytes)

              ram+csiz:  start of program code
                         ... (program code) ...
                xxxx-1:  end of program code

                  xxxx:  external variable area (y bytes long)
                         ... (external data) ...

                xxxx+y:  free memory,
                                available for
                                        storage
                                            allocation

                  ????:  as low as the machine stack ever gets
                                local data, function parameters,
        machine stack:          intermediate expression results,
                                etc. etc.
high memory:      bdos:  machine stack top (grows down)
--------------------------------------------------------------------
```

Note that "xxxx" is the first location following the program code and "y" is the amount of memory needed for external variables.

To implement overlays, the first thing necessary is to decide just where the swapped-in code is to reside. Earlier versions of BDS C had local data frames growing up from low memory, starting from where the externals ended, making it difficult to determine the lowest memory location safe to swap into. The scheme suggested then for handling overlays was to leave sufficient room between the end of the root segment code (the root segment contains the "main" function and run-time package; it loads at the start of the TPA, always remains in memory, and controls the top level of overlay swapping) and start of the external data area to accommodate the largest possible swapped-in segment combination. This is still a viable scheme for version 1.4; here is the modified memory map, accommodating this first method of handling overlays:

```
low memory:  base+100h:  C.CCC run-time utility package (csiz bytes)
               ram+csiz:  start of root segment code
                          ... (root segment code) ...
                 zzzz-1:  end of root segment code

                   zzzz:  start of overlay area
                          ... (overlay area) ...
                 xxxx-1:  end of overlay area

                   xxxx:  external variable area (y bytes long)
                          ... (external data) ...

                 xxxx+y:  free memory,
                                 available for
                                         storage
                                             allocation

                   ????:  as low as the machine stack ever gets
                                 local data, function parameters,
         machine stack:          intermediate expression results,
                                 etc. etc.
high memory:       bdos:  machine stack top (grows down)
```

Note that "zzzz" is where segments get swapped in, guaranteed that the longest segment doesn't reach "xxxx".

With version 1.4, it is just as feasible to put the overlay area AFTER the externals. memory map for this alternative configuration would be:

```
low memory:  base+100h:  C.CCC run-time utility package (csiz bytes)
               ram+csiz:  start of root segment code
                          ... (root segment code) ...
                 xxxx-1:  end of root segment code

                   xxxx:  external variable area (y bytes long)
                          ... (external data) ...
               xxxx+y-1:  end of external data area

                 xxxx+y:  start of overlay area (ssss bytes long)
                          ... (overlay area) ...
          xxxx+y+ssss-1:  end of overlay area

            xxxx+y+ssss:  <unused memory>

                   ????:  as low as the machine stack ever gets
                                 local data, function parameters,
         machine stack:          intermediate expression results,
                                 etc. etc.
high memory:       bdos:  machine stack top (grows down)
```

If you plan to use the storage allocation functions (alloc, free, sbrk, rsvstk) in y program, then this second scheme would require you to call the "sbrk" function with argument "ssss" (the size of the overlay area) since, by default, storage allocation always begins with the area immediately following the end of the externals. For the remainder of this document, I will assume the FIRST of the above two schemes is being used.

OK, with the generalities out of the way, let me say something about just how to create "root" segments and "swappable" segments with BDS C. First of all, we would like all functions defined within the root segment to be accessible by the swapped segment(s)...this is accomplished by causing CLINK to write out a symbol table file (containing all function addresses) to disk when the root segment is linked. The -w option to CLINK will do the trick; this symbol table will be used later when linking the swappable segments.

When linking the root segment, use the -e option to set the external data area location; keep in mind that there must be enough room below the externals to hold the largest swapped-in segment at run time (I'm using the term "below" in the sense that low memory is "below" high memory; graphically, in the preceding memory maps, "below" means toward the top of the page.) If the -e option is omitted, CLINK will assume the external data starts immediately after the end of the root segment code; this is OK only if you're using the SECOND scheme.

Within the code of the root segment, then, a swappable segment is loaded into memory from disk by saying:

```
        swapin(name,addr);    /* read in a segment..don't run it */
```

where "addr" is the location following the last byte of root segment code (for the first scheme.) You can find this value by linking the root once without giving the -e option and reading the -s statistics written to the console after the linkage. To actually execute the segment, you have to call it indirectly using a pointer-to-function variable.

Here is an example. We'll declare a pointer-to-function variable called "ptrfn", swap in a segment named "foo" at location 3000h, and call the segment. The sequence would look like this:

```
        int (*ptrfn)();          /* can be whatever type you like */
        ptrfn = 0x3000;
        ...
        if (swapin("foo",0x3000) != -1) /* check for load error */
             (*ptrfn)(args...);         /* if none, call the segment */
        ...
```

The "swapin" routine returns -1 when a load error occurs. Note that the swapped-in code might not return any value, but the pointer-to-function must be declared with SOME kind of type. Use "int" if nothing else comes to mind. When a segment is invoked, as above, control passes to the segment's "main" function. There is no reason at all to require args to be of the "argc" and "argv" form; there is nothing special about a "main" function other than the property it has of getting called first. The "main" function within the swapped-in segment is the ONLY allowed entry point for the segment.

A simple "swapin" function is given in STDLIB2.C. It can be made shorter by skipping all the error testing, or can be expanded to detect an attempted load over the external data area by comparing the last address loaded with the contents of location ram+115h...if you've never done any low-level hackery, you get the value of the 16-bit address at location ram+115h by using indirection on a pointer-to-integer (or -unsigned.) Note that location RAM+115h ALWAYS contains the address of the base of the external data area.

Now we know how to do everything except actually create a swappable segment.

K, a swappable segment is basically just a normal C program, having a "main" function just ike the root segment, except that the C.CCC run-time utility package is NOT tacked on to the ront of a swappable segment (the C.CCC in the root segment will be shared by everyone.) The ther difference between a swappable segment and the root segment is the load address; while e root segment always loads at ram+100h (where "ram" is 0 for standard CP/M, or 4200h for the odified" CP/M), a swappable segment may be made to load anywhere. Once you've compiled the wappable segment, you give a special form of the CLINK command to link it:

```
        A>clink segmentname -v -l xxxx -y symbolfile [-s ...] <cr>
```

here "segmentname" is the name of the CRL file containing the segment, "-v" indicates to CLINK hat a swappable segment is to be created (so that C.CCC is not attached), and "-l xxxx" letter ell followed by a hex address) indicates the load address for the segment.

ince you'll probably want to yank in the symbol file created by the linkage of the root egment, use the -y option to do so. If you don't, then CLINK will yank in fresh copies of unctions like "PRINTF" and "FOPEN", etc., even if they have already been linked into the root egment. It would be a waste to have multiple copies of those memory hogs in there at the same ime! By reading in the symbol table from the root segment, it is insured that any routines lready linked in the root will be made available to the swapped-in segment. The root segment, hough, cannot know about functions belonging to the swapped-in segment through the use of a ymbol table. That would require some kind of mutually referential linking system beyond the cope of this package.

h well. When linking the segment, you may specify -s to generate a stat map on the console, nd -w to write out an augmented symbol table containing not only the symbols read in from the oot segment's symbol file, but also the swappable segment's own symbols. This new symbol file ay then be used on another level of swapping, should that be desired.

xample: (The addresses given in this example are for a RAM at 0000h CP/M; if you have the odified 4200h CP/M, fudge accordingly.)

et's say you've got a program ROOT.C, which will swap in and execute SEG1.C and then overlay G1.C with SEG2.C. ROOT.COM loads at 100h and ends, say, before 3000h. We'll load in the egments at 3000h, and set the base of the external data area to 5000h (this assumes neither egment is longer than 2000h.)

he linkage of ROOT would be:

```
        A>clink root -e 5000 -w -s <cr>
```

his tells CLINK that ROOT.COM is to be a root segment (no "-v" option used), the externals start at 5000h, a symbol file called ROOT.SYM is to be written, and a statistics summary is to e printed to the console.

he linkage of each segment would appear as:

```
        A>clink segl -v -l 3000  -y root -s -o segl. <cr>
```

he command line tells CLINK that SEG1.COM is to be a swappable segment (the "-v" option) to oad at location 3000h, the symbol file named ROOT.SYM should be scanned for pre-defined unction addresses, a statistics summary should be printed after the linkage, and the object ile is to be written out as SEG1 (as opposed to SEG1.COM, to avoid accidentally invoking it as CP/M command.)