Solvin Patent

PHILIPS

PHILIPS INTERACTIVE MEDIA

Technical Note #70.1

Inhibiting the Replication of Global Data

Charles Golvin
Supersedes TN# 70

October 25, 1991
Revised August 20, 1992

This technical note describes a method of inhibiting the double memory cost that applications pay for initialized data, initialized data references, and initialized code references.

Copyright © 1992 Philips Interactive Media of America.
All rights reserved.
This document is not to be duplicated or distributed without written permission from Philips Interactive Media of America.

Inhibiting the Replication of Global Data

Background

One advantage that OS-9 claims is the use of reentrant code. This represents a memory savings in many situations. For example, on a multi-user OS-9 machine many users could be simultaneously using the same copy of **dir** so that the overhead is only in the data and stack areas for each process. The one requirement for each process is that it be able to obtain pristine copies of the initialized data, data references, and code references that **dir** requires to operate correctly. OS-9 is able to guarantee this capability, because it contains the initialized data within the code module. When a process activates the code, a local copy of the initialized data is moved into the process's data area and properly initialized (this work is actually performed by the two system calls, **F$Fork** and **F$Chain**, that activate new processes).

This is a good scheme for applications that are truly reentrant, such as **dir**. However, in general, the code in CD-I applications is not used by more than one process at a time. Thus, this replication can mean that the application must pay twice in memory for each byte of initialized data. For the purposes of this note, I will refer to the amalgam of initialized data, data references, and code references as "global data."

The Strategy

The approach chosen for removing the second copy of initialized data consists of two pieces: a build time side and a run time side. Since the second copy of the global data is in the local area of the application (and is, hence, required), the only opportunity for memory reduction is to break the bond between the code module and the copy of global data that is normally concatenated to it by the linker.

Essentially, the build time side splits the application module into a code module and a data module. The code module created is a copy of the original program, with the exception of the global data and the module header offset to the initialized references. The original global data is copied into the output data module, and it is replaced in the code module by a short string of null bytes. Figure 1 below illustrates this process.

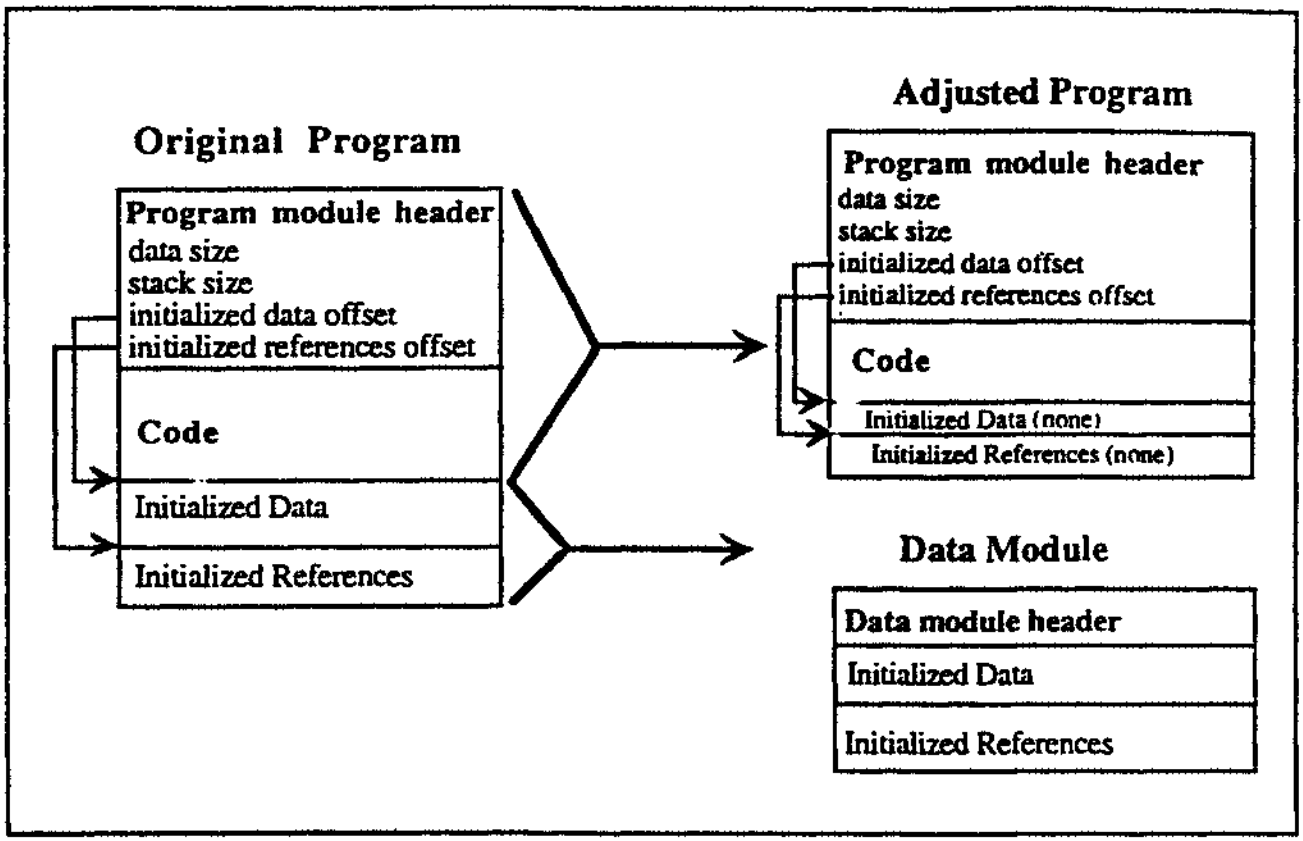

Figure 1: Copying of Global Data into the Output Data Module

The run time side is a modified version of **cstart.r** that performs some of the functions normally completed by **F$Fork** or **F$Chain**. Rather than retrieving the global data from the code module, this version of **cstart.r** loads the data module that was created by the build time side from the current execution directory. The global data is copied to the local data space and properly initialized; the data module is then removed from memory.

It is worth noting that programs using this method remain reentrant. The only added requirement is that the forking application must have its execution directory set to the directory containing the data module created by the build time program. For most CD-I applications, this means the **cd** device.

How It Actually Works

When an application is activated by OS-9 (either via **F$Fork** or **F$Chain**), the operating system call examines the module header to determine how much memory must be allocated for local data and stack space. (In general, "don't care" system memory requests come from the least used plane and are allocated on a first fit basis, starting from high memory. However, the allocation of the local data and stack area for a process is a special case. These memory requests come from the least-used plane, but start at low memory.) After allocating this memory, the system determines from the module how many bytes of initialized data need be copied. It

copies this data from the module into the local memory space of the application; the system then adjusts data and code references based on the memory addresses of the local data space and code module, respectively. Once this action is completed, the application begins executing its code. Figure 2 below illustrates what happens when an application is activated by OS-9.

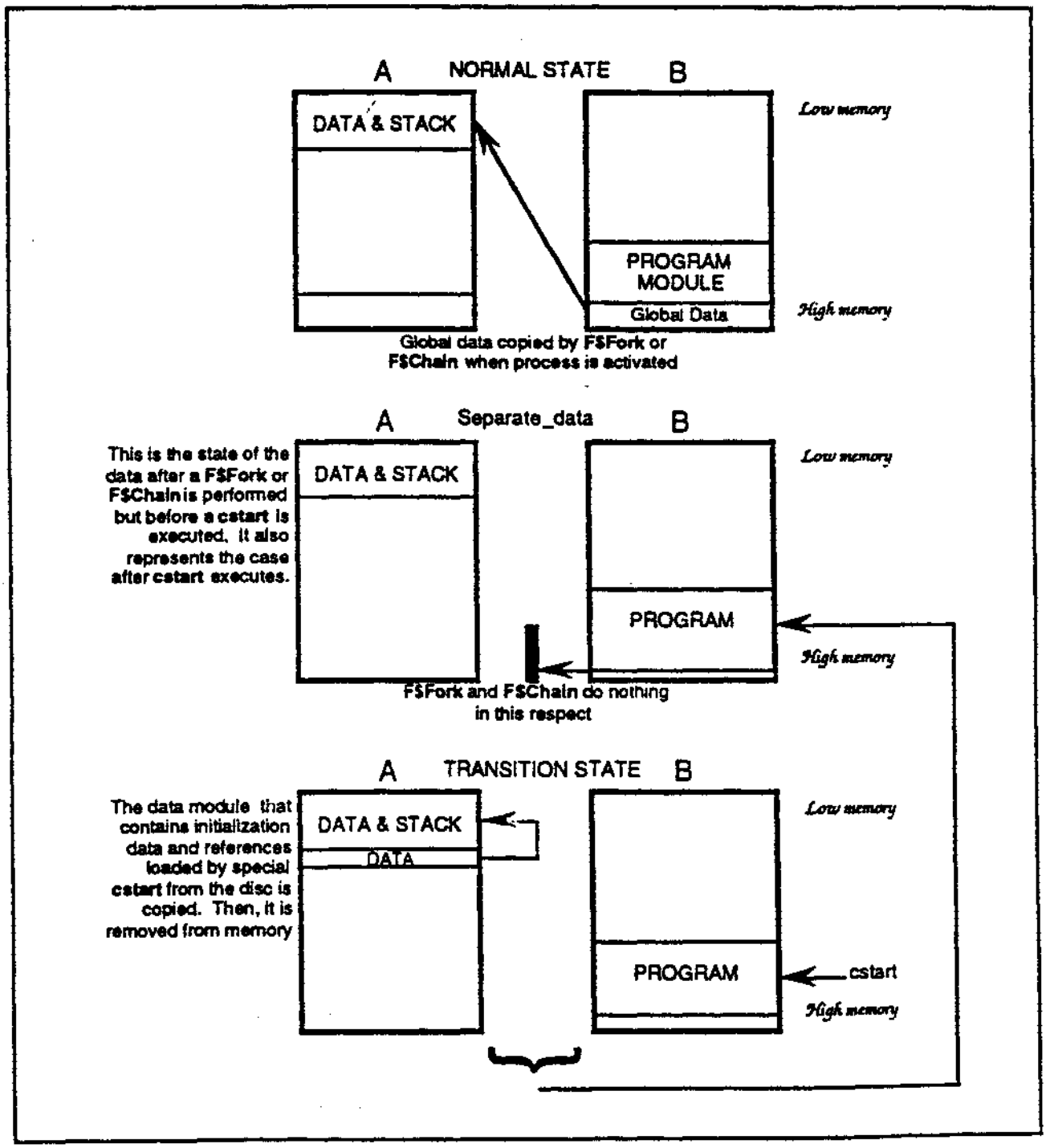

Figure 2: How It Works

When a program has been modified via the method described above, the data and stack requirements remain unaltered so that memory is allocated as described above. When the system examines the module, it finds that there are no bytes of global data to copy, no initialized data references to adjust, and no initialized code references to adjust. It then begins executing the application code, beginning with cstart.

The modified cstart uses **F$Load** to load the data module into memory. It imposes a naming convention on the data module: the same name as the application with **_data** appended at the end. Fortunately, the build time program defaults to this name when it names the data module. It then uses **F$Link** to find the address of the application module in memory and determines the location of the local data space (contained in one of the program's address registers). This provides all the information needed to correctly make a properly adjusted copy of the global data. Finally, it calls **F$UnLink** to restore the application's original link count and **F$UnLoad** to remove the data module from memory. (Recall that this is the purpose of the whole exercise.)

How To Use It

The build time program is called **separate_data**, and there are versions for Sun3, Sun4, and Macintosh machines. The run time piece is also called **cstart.r**, although it can be renamed at your whim. Version 3.2.3 of the Microware cross-compiler supports a **-CS=<startup>** option that allows the version of **cstart.r** to be specified on the command line. Those using earlier versions of the cross-compiler can either rename the old version of cstart.r and copy the new version to the /dd/lib area, or copy the contents of /dd/lib to a new directory along with the new version of cstart.r and make use of the CLIB environmental variable.

The following is a portion of a UNIX makefile that uses version 3.2.3 of the cross-compiler and the technique described above to create an OS-9 application.

```
cdi_fred:   src/fred.c
      cc68  -CS=rels/cstart.r  src/fred.c  -f=bin/cdi_fred      # compile and link
      cp  bin/cdi_fred  bin/cdi_fred.bak                       # make backup copy
      separate_data  bin/cdi_fred                        # do the splitting thing
```

The result of this process is a modified executable module called **cdi_fred** and a data module called **cdi_fred_data** (both located in the bin directory). The following is an excerpt from the master script used to build a disc that uses the above-created **cdi_fred** as the start-up application.

```
      application  file  freddy  from  "/home/fred/bin/cdi_fred"
      yellow  file  freds_data  from  "/home/fred/bin/cdi_fred_data"
      ...
      "cdi_fred"  protection  0x555  from  freddy
      "cdi_fred_data"  protection  0x555  from  freds_data
      ...
```

Pluses and Minuses

The main benefit from this technique is, of course, the elimination of an extra copy of the global data needed for the program to run. This memory always comes out of the plane in which the application is loaded. To get an idea of the savings you can reap, run **ident** on your application and subtract the **Init. data off** field from the **Module size** field.

The main disadvantage of this technique is the second disc access required when the **FSLoad** call is made to load the data module. Bear in mind that the time to read this data off the disc is insignificant, because it would have to be read in either case. One other possible drawback is the potential growth of the module directory table by virtue of loading a new module into memory—this is player dependent because each player may contain a variable number of initial modules at startup.