



## ***smxDLM***<sup>TM</sup>

### ***Dynamic Load Module Support***

***smxDLM permits independent executable modules (dynamic load modules) to be loaded and executed as tasks under smx. This gives additional flexibility since it permits parts of an application to be separate from the main application.***

### **Operation**

*smx* is designed to be linked with the application code into a single executable. While ideal for most embedded applications, this is too limiting for some.

*smxDLM* extends *smx* to permit an application to be divided into a resident section, referred to as the *resident* and one or more transient sections, referred to as *Dynamic Load Modules* (DLM's). DLM's are not linked with *smx*. Instead, each DLM is linked with a pseudo *smx* library that uses software interrupts, or traps, to invoke *smx* functions. The resident includes the standard *smx* functions, the DLM loader, resident application code, and possibly other SMX products.

DLM's may be loaded and unloaded, at will, and any number may run simultaneously.

The resident is likely to be linked, located, and burned into ROM or flash. However, DLM's are most likely to be linked into relocatable formats and loaded into RAM from disk or downloaded from another computer.

DLM's are an ideal way to add functionality or to distribute bug fixes to products which have already been shipped. To facilitate this, the resident should be as minimal as possible and thoroughly debugged. (It is basically a custom os.) Complex or

### **Features**

- Dynamically loads independent executable modules as tasks.
- Saves memory by allowing tasks to be loaded only as needed and unloaded when not needed.
- Provides customer programmability via adding custom DLM's using low-cost end user DLM Kit.
- Permits distributing *smx* in ROM.
- Permits downloading new or altered tasks to remote systems.
- Solves library reentrancy problems by permitting multiple copies of libraries.
- Enforces greater subsystem independence.
- Allows programmers on the same project to work more independently.
- Loader for MZ, NE, or PE file formats is included
- Simple API for loading and unloading DLM's
- *smx* call mechanism can be easily extended to allow calling other functions in the resident from DLM's.

likely-to-change portions of the application should be implemented as DLM's. These can be altered and easily reloaded in the future.

### **How *smx* Calls Work from DLM's**

DLM application code differs in only minor ways from resident *smx* application code. (In fact, DLM's are most easily debugged by initially linking them with the resident.)

Prior to loading, each DLM is linked with a special, pseudo-*smx* library (and any other libraries it requires). This library has a pseudo call of the same name and syntax as each *smx* call. Each pseudo call loads its call ID into a register, then executes a software interrupt or trap. The interrupt or trap level is user selectable.

In the resident application, the *smx* interface routine interprets the call ID, adjusts the stack, and invokes the specified *smx* call. When done, the *smx* call returns directly to the DLM caller and passes back a return value, as normal. Thus, the operation is transparent to the DLM application code — it need not know which *smx* library (actual or pseudo) it has been linked to.

## Loading DLM's

DLM's are typically linked as .exe files that are loaded from disk. The *smxDLM* loader currently supports the MZ, NE, and PE file formats for x86 systems. Support for other re-locatable formats will be added in the future. DLM's can also be in absolute format (i.e. located code) and be loaded into fixed locations. This tends to undermine their usefulness, but may be the only option in some cases.

*smxDLM* code is supplied to load from disk using *smxFile*. Intermediary routines are provided for the file i/o calls used by the loader. These few calls can be re-implemented to get the data from another source, such as a data link or network.

The DLM loader obtains memory from the *smx* heap, loads the DLM into it, performs segment or offset fixups, and creates a *root* task for the DLM startup code. The loader then starts the DLM's root task. This causes the DLM's startup code to begin executing.

The API for use by the resident to load and unload DLM's consists of two routines: *create\_process()* and *delete\_process()*. They are used as follows:

To load and start a DLM:

```
PRCB_PTR process;

process = create_process(path, pri, stack_sz);
if (process)
    start(process->tcb_handle);
else
    /* DLM load failed */
```

To unload a DLM:

```
if (!delete_process(process))
    /* DLM unload failed */
```

## DLM Operation

Because handles are assigned values by the linker and because a DLM is independently linked, the DLM does not know the values of handles in the resident or in other DLM's. Suppose that a DLM wants to send a message to XchgA in the resident. To do so, it must first get the value of the handle from the handle table, then do the send:

```
xchga = get_handle ("XchgA");
send (msg, xchga);
```

The handle table is maintained by *smx* in the resident. The above code sequence will work correctly in both a free-standing DLM and in a DLM linked to the resident during debugging.

Once a DLM begins operation, it can spawn any number of additional tasks and create any number of other *smx* objects. As it creates objects, the DLM may register them in the handle table so that the resident and other DLM's can access them. This is done the same as in normal code:

```
taska = create_taskf(taska_main, 3, 500);
BUILD_HT(taska, "TaskA");
```

## Other Information

Full source code is provided for *smxDLM*. This makes it easy to add non-*smx* services, such as file, GUI, stack, and special resident application services. It is also possible to alter or change the loader.