TransSkel Programmer's Notes

5: `SkelInit()` and `SkelInitParams`

Who to blame: Paul DuBois, dubois@primate.wisc.edu
Note creation date: 10/09/93
Note revision: 1.01
Last revision date: 01/03/94
TransSkel release: 3.04

This Note describes the `SkelInitParams` structure used by `SkelInit()` as of TransSkel release 3.04.

Purpose and History of `SkelInit()`

The first TransSkel function an application calls is usually `SkelInit()`, which initializes the various Macintosh managers and determines some of the characteristics of the machine on which the application is running.

In the earliest releases of TransSkel, `SkelInit()` took no arguments:

```
void SkelInit(void);
```

This was somewhat limiting. For instance, an application may wish to call `MoreMasters()` to preallocate master blocks of memory pointers, and it may wish to install a growzone procedure for dealing with low-memory conditions. `SkelInit()` is the most logical place to do these things, so it was changed (in release 2.00) to take two arguments:

```
void SkelInit(Integer noMasters, GrowZoneProc gzProc);
```

Here `noMasters` is the number of times to call `MoreMasters()` and `gzProc` is the growzone procedure. This change introduced an incompatibility with previous releases of TransSkel and applications had to be changed accordingly.

The new `SkelInit()` was more capable, but it left other limitations unaddressed. `SkelInit()` calls `InitDialogs()`, which can be passed a pointer to a procedure to be called if a fatal system error occurs. However, `SkelInit()` just passed `nil` to `InitDialogs()`, leaving the application no way to specify the procedure if desired. `SkelInit()` also provided no way to adjust the application stack size.

It's possible to rectify these problems by adding more arguments to `SkelInit()`, but that's another incompatible change. Furthermore, who knows what changes might be desireable in the future, necessitating yet more arguments, and requiring applications to change once more the way they call `SkelInit()`?

The New Calling Sequence

If an incompatible change is going to be made, it might as well be such as to limit the impact of additional changes in the future. The approach chosen was to modify `SkelInit()` to take a *single* argument, a pointer to a structure specifying the initialization parameters. This allows the calling sequence to remain the same forever. The structure itself may change someday, but that will cause little problem even if the structure itself changes someday, as will be explained shortly.

The structure is declared like this:

```
typedef struct SkelInitParams SkelInitParams, *SkelInitParamsPtr;

struct SkelInitParams
{
      Integer           skelMoreMasters;
      GrowZoneProcPtr   skelGzProc;
      ResumeProcPtr     skelResumeProc;
      Size              skelStackAdjust;
};
```

This structure allows the application to specify a resume procedure and the amount by which to adjust the default stack size. As of release 3.04, `SkelInit()` is declared like this:

```
void SkelInit (SkelInitParamsPtr p);
```

To make it easy for applications requiring no special setup, default parameter values are used if the application passes `nil`.

This change is incompatible with releases of TransSkel prior to 3.04, and older applications will need to change the `SkelInit()` call. However, once this is done, any future modifications should have little impact. The `SkelInit()` calling sequence will remain stable: one argument, a pointer to a structure. And it will continue to be true that an application wishing to use the default initialization parameters can pass `nil` to `SkelInit()`.

Future changes to the `SkelInitParams` structure have the potential to cause problems. Suppose you have a C program that specifies non-default initialization values, as follows:

```
SkelInitParams initParams =
{
      10,
      MyGrowZoneProc,
      MyResumeProc,
      8096
};

SkelInit (&initParams);
```

Now suppose that in a future release of TransSkel, `SkelInit()` is changed in a way that necessitates new initialization parameters. In C this is not a problem if the new parameters are added to the end of the `SkelInitParams` structure and zero is the default value for each of them. Since C initializes unspecified structure elements to zero, the program can simply be recompiled. The compiler will supply the new fields for `initParams` and initialize them to zero automatically. Thus, changes to the structure do not automatically necessitate changes to the application source.

But that won’t work if any new parameter has a non-zero default. Nor will it work if/when TransSkel 3 is ported to Pascal, since Pascal doesn’t initialize structures the same way. To handle this, a new call `SkelGetInitParams()` is introduced. You can pass it a pointer to a

`SkelInitParams` structure. The structure will be filled in with the default values. Then you can change any fields for which the defaults are unsuitable and pass the structure to `SkelInit()`:

```
SkelInitParams initParams;

SkelGetInitParams (&initParams);    /* get default values */
initParams.skelMoreMasters = 10;    /* change fields appropriately */
initParams.skelGzProc = MyGrowZoneProc;
initParams.skelResumeProc = MyResumeProc;
initParams.skelStackAdjust= 8096;
SkelInit (&initParams);
```

This method will work for either language and makes sure you get default values for any structure elements you don't change, whatever those defaults happen to be.

Summary

If your application requires no non-default initialization parameters, call `SkelInit()` like this:

```
SkelInit (nil);
```

Otherwise, declare a structure of type `SkelInitParams`, pass it to `SkelGetInitParams()`, change the fields appropriately, and pass the structure to `SkelInit()`:

```
SkelInitParams initParams;

SkelGetInitParams (&initParams);
/* ...change fields here... */
SkelInit (&initParams);
```