Changes in the TSP++ class library from v2.0 to v3.0

This document describes the internal changes in the JulMar TSP++ class library from version 2.0/2.1 to version 3.0. The reason for these changes is two-fold.

First, this release of the class library has been heavily modified for *third-party* call control. This is where the service provider is controlling a large number of resources, such as a PBX switch or an ACD.

Second, due to increased demand for this library JulMar has approached various companies to determine the feasibility of selling the library to a larger company that can adequately support it.

Important Note:

> Because of the changes, this latest version of TSP++ is not source-code compatible with previous releases and will require some relatively major rework of existing providers developed under the 1.x or 2.x products.

Removal of the Microsoft Foundation Classes

The largest change consists of removing the use of MFC throughout the main library. In previous versions, MFC was used for various tasks:

1. List and Array management of objects
2. Dynamic (RTTI) creation of objects and overriding of objects outside the library.
3. String management
4. 16-bit to 32-bit compatibility.

List and Array management of objects

The TSP++ library controls and maintains several objects representing telephony devices and concepts. Many of these objects must be stored in growable arrays or linked-lists. Every object in the library was derived from MFC's **CObject** class and therefore could be stored into the list, array and map objects present in MFC.

This management was replaced in v3.0 with STL (Standard Template Library) based objects. Using templates allows various object-specific arrays, lists and maps to be generated during compile time. In this version of the library there is no "base" object which everything is derived from. The following elements from STL were used to replace MFCs collection classes:

MFC Collection	Basic STL template
CMapDWordToString	**Map**
CMapDWordToOb	**Map**
CObList	**List**
CObArray	**Vector**
CPtrArray	**Vector**
CFlagArray	**Bitset**

Due to the above change in collection management, all the linear array traversal functions now use STL *iterators* to walk through vectors, and take advantage of various algorithms when appropriate (such as **find**).

Dynamic (RTTI) creation of objects and overriding of objects outside the library

Another primary usage of MFC was to dynamically create objects of unknown types within the TSP++ library itself. This feature allows the TSP++ library to control the lifetime and scope of almost all of the telephony objects internally but still allows a developer to override any of the basic objects using standard C++ derivation.

This was accomplished in MFC using the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** macros which add a primitive form of run-time type information (RTTI) to each of the objects.

The ANSI standards define a C++ method for RTTI that goes beyond the MFC implementation, but it unfortunately does not allow for the creation of a derived object type without knowledge of the constructor and object layout.

The necessary support was added directly into the TSP++ library through the creation of a factory-management object that has knowledge of each of the used objects in the developed service provider. It requires that the developer add a single "registration" line (similar to the **IMPLEMENT_DYNCREATE** macro which MFC uses for the same purpose) to one of their source files so that the proper object may then be created at run-time. The factory utilizes the built-in C++ RTTI support to identify the object that needs to be created and then instantiates it by calling the appropriate constructor.

String management

As of version 2.0 of TAPI, all strings coming into and out of the provider are required to be in UNICODE. The TSP++ library allows the developer to work with either straight UNICODE (for Windows NT) or with ANSI (for Windows 95). Two separate versions of the 32-bit library are shipped with TSP++. The first assumes that all the string management in the service provider is UNICODE and therefore it does no translation on strings. The second assumes that all strings should be in ANSI form and automatically converts the strings on entry and exit.

To encapsulate the difference between the two libraries and allow a single source base to work with either, the TCHAR.H header is used, and MFCs **CString** class was used in all places where strings were passed through a service provider. Using the **CString** object allowed the library to automatically manage the memory allocation and free for each string reference and to automatically handle UNICODE throughout the library.. In addition, with VC 4.x, the strings are reference counted and therefore conserve memory.

With the removal of MFC, the **CString** class has been replaced with the STL **basic_string** template. This is typedef'd in the headers to the namespace **TString** which is either derived from **char** or **wchar_t** depending on whether UNICODE is used or not. This step is necessary since the UNICODE and ANSI versions of the object are difference (unlike the **CString** class).

16-bit to 32-bit compatibility

The previous releases of TSP++ have always maintained a "backwards-compatible" notion so that upgrading to the new library was not a nightmare of changes. As part of this, MFC was used to allow the same source code to be compiled as a 32-bit TSP or 16-bit TSP (for Windows 95 or 3.1). Since MFC is available on both platforms, it provided an easy migration path for all the users.

This feature of the previous versions has been lost. Since templates are not supported in 16-bit C++, the new STL implementation cannot be compiled into a 16-bit TSP.

With the latest release of TAPI (2.1) however, 32-bit service providers may now be used on all current Windows platforms. This indicates that 16-bit and 32-bit compatibility is no longer a necessary design goal and therefore has been dropped with this release.

Any party needing to continue development of a 16-bit service provider can purchase and use the 2.x version of the library.

Multi-Threading Support

Thread synchronization and "race" conditions

The first 32-bit version of TSP++ added thread-synchronization support into the library. This support was not heavily tested during the 2.0 port primarily due to the lack of high quality multi-threaded TAPI applications.

With TSP++ 2.1, several in-house "dirty" multi-threaded applications were developed to test various scenarios involving a TSP bult with TSP++. The holes uncovered in the testing were fixed and released with TSP++ 2.1. The two main problems found were:

1. Some scenarios were identified where a HDRVCALL handle could be deleted and then passed back through on another function call while the library was de-allocating the structure associated with the TAPI call handle causing an invalid pointer reference.

   This was especially prevalent when multiple operations were being performed simultaneously against a call and the call was dropped/deleted. Since the TSP++ library maintains a prioritized list of requests, and a drop request is the highest priority, it could occasionally catch another worker thread by surprise when the object disappeared from under it.

   This problem was solved by reference counting the call object with respect to each request that has a pointer to it. In that fashion, the call object is guaranteed to be in existence in the TSP++ library while any outstanding request object is still pending action (whether that is to run or to be canceled due to a drop/close). In this fashion, the call is not immediately deleted on the **lineDeallocateCall**, and instead is kept around until all references internally are completed.

2. When high activity was being placed against a particular address, there was a small window where two calls could be created through different **lineXX** methods (i.e. **lineMakeCall** vs. **lineUnpark**) when the **LINEADDRESSCAPS** specified that only one call could be active.

   This scenario was found by having over 30 running TAPI applications attempting to place calls simultaneously. Since the library only incremented the **dwActive** field once a request was actually created, depending on where the context switch occurred, two threads could get a request queued, both believing that there was no active call on the address (which is true at the moment that the request was initiated).

   This problem was solved by having the line methods that can create a new call check in the queue for existing requests which will impact the call counter before allowing the call to succeed.

Various other minor modifications were made with v2.1 to streamline thread access and to minimize thread contention.

Object Synchronization

As with any multi-threaded product, there are places where data structures (such as lists or arrays) are changed or queried that requires some thread-synchronization support to be applied. In the case of the TSP++ library, Critical Sections are used to prevent accessing data within objects when the data is changing or volatile.

Version 2.x of the library associated a Win32 Critical Section with every telephony object (line, phone, address, and call). This critical section was then tagged by a thread when it was about to change or read sensitive information within the object in question.

After some research into the implementation of the Win32 Critical Section, it was decided that this approach was too expensive when the object count grew over 100 or so (such as third-party).

The basic problem stems from the fact that even though the critical section is contained within the user space, it uses a kernel event object (per critical section) to put waiting threads to sleep. When the critical section is released by the owner thread, that event is used to wake up the first waiting thread.

While the creation of 100 kernel objects is not noticeable on most Win32 systems, the creation of a 1000 or more is very noticeable – especially on Windows 95 where they appear to be maintained within some kind of linked list.

So the question becomes "How do we synchronize access to our objects without using a kernel object per access?"

Several solutions were discussed and prototyped to determine their feasibility. The basic solutions discussed were:

1. Have a single "global" critical section which is used whenever a read/write to sensitive data (such as array changes, etc).
2. Have a critical section per line (vs. per object) and have the address/call objects look up to the line for protection.
3. Create a multi-reader/single writer gate object.
4. Create a non-kernel critical section using atomic operations and continue to associate one per object.

Single "Global" critical section

While appealing, this approach leads to a serious bottle-neck in performance. It was brought up initially and immediately discarded due to the obvious problems associated with "gate" style solutions such as this.

Critical Section per line

This approach seems more appealing that #1, but when it is considered that a single server-style service provider could have over 1000 lines (actually 65000 in our current model) it starts to have the same problems as the implementation used in the v2.x library (i.e. too many kernel objects).

In addition, gating changes to call appearances via the line owner presents the same performance issues related to solution #1.

Create a "multi-reader/single-writer" gate object

This solution involves creating a standard serialization object which allows for multiple reader threads to access the object simultaneously but stops all threads accessing the object while it is being written to. There were two main problems that came out of the discussion for this solution. The first was that identification of read vs. write events in the library would be a huge task. The second and much more major problem deals with inter-object dependencies.

Many of the objects within the TSP++ library have internal dependencies upon each other. For example, a function in the line object could call a function in one of its address objects, which could call a function in one of the call objects. With the multi-reader single writer gate, scenarios quickly come up where you deadlock the thread because it has a write lock in the line object and needs a read lock against the same line while in the call object. Or, the thread may need to modify line data while in the call object, but the thread already has a read-lock from a previous function in the stack flow.

It quickly became apparent that trying to isolate read events vs. write events would be a nightmare since depending on different variables and function which needs a read event can quickly become a function that requires write access. So, this solution was discarded.

Create a non-kernel critical section using atomic operations

This solution was the most appealing, first because it required the least amount of rework in the library. By having a synchronization object per telephony object, we maximize the throughput of thread activity. Since data contained within an object is protected by the object itself, we don't have to go outside the bounds of the object for access.

The major problem that came out of this solution was how to block the waiting thread(s) and then wake them back up again when the object is released. One obvious solution is to use a *spin-lock*, where the thread loops waiting for the resource to be available. The problem is that this eats CPU cycles in a big way.

The time that that any given thread has the object locked is our critical time. Depending on how long any thread must wait determines how long that thread would *spin* in the loop (and thereby waste CPU cycles). The threads could be put to sleep using the Win32 **Sleep** function, but then it starts to impair on our performance to some extent.

Through analysis of code flow, it was determined that an object remains locked for at most 1 second. In most cases it was only a few clock cycles. In many cases, there is no thread contention at all – the object is released so quickly that other threads never wait for it. There are some cases however where multiple threads can be waiting for an object release. This multiples the wasted CPU problem and adds an additional issue: it would be preferred to have the first waiting thread access the resource when it is released. Using a spin-lock doesn't allow for this however, since the first thread which gets CPU time will get the resource. Obviously additional information could be tracked, such as a priority queue of thread-ids waiting, etc. Adding all that increases our overhead however, and this synchronization object is used for *every* telephony object in the system, so the smaller and faster the better.

Another suggested solution was to use a kernel event *only* when multiple threads are pending for an object. In other words, the creation of the event is held until it is determined that it is really required. Once a thread is required to wait for an object, it creates a Win32 event and blocks itself on the event. When the resource is released, the releasing thread looks to see if an event has been created and then sets it, thereby releasing the first waiting thread.

This solution solves the "spin" problem presented earlier since the threads are truly put to sleep, but if all of the lines eventually had some thread contention we are back where we started. A solution to *that* problem is to delete the event when it is no longer needed. Potential performance problems were discussed with this approach, but upon timings of process-local event creations under Windows NT and Windows 95 it was determined that this was probably not an issue.

Two separate solutions have been coded and are awaiting full testing. The first implements a critical section using atomic operations and a Win32 event object when multiple threads block against the resource. The event is then destroyed after a certain timeout of non-thread contention against the object (i.e. where the processing thread *always* got immediate access to the resource).

The second solution is like the first but adds a spin lock for a certain timeout to determine if we can keep our kernel object creations down to a minimum without sacrificing overly CPU cycles.

Interface-wise both objects are identical. A current project will allow for full testing of both implementations to determine which holds the best performance vs. CPU usage.

Third Party Call Support

TSP++ Version 3.0 is the first version of the library targeted at building large-scale third-party service providers. As such, it has some optimizations and enhancements that target that environment. These changes are equally applicable to the first-party environment but aren't really needed due to the lower number of objects.

Searching and Arrays

The first major difference is that many of the arrays used throughout the library are no longer linearly searched for information. Instead, hash tables are built up front and kept in synch with the array to quickly identify an object based on some criteria. This increases the memory footprint required (by about 1-200k), but since this is really targeted for a server, it was decided that it was not really an issue.

An example of this is the call object list. The call list is maintained within each individual address object, which is contained by each line object. On most PBX and ACD systems, a call is represented in the switch by a *call-identifier* (callid) which is associated with the call object. In previous versions you could only find a call by call-id through a linear search of the address it is contained on. In this version, there is a global function that will locate a call by identifier across all lines and addresses in using a map.

Line objects may also be associated with a key that may be later used to quickly find them. The key is determined by the developer writing the provider and would be specific to the implementation.

Other arrays and lists use the STL find() algorithms to locate objects within the appropriate collection. An example is the call completion identifiers.

User Interface Library

TSP++ Version 3.0 splits out the TSPUI interface into a separate library so that a separate UI dll may be created. In previous versions (to maintain compatibility with 16-bit versions) the UI code was integrated into the TSP++ library itself.

The UI library allows for the user interface components to be created in MFC or using SDK code. It has helper functions to convert line device id's into provider id and to check whether a provider is installed in the system yet (functions which used to be integrated into TSP++).

Agent Support

TSP++ Version 3.0 adds support for the creation of an agent proxy application that is tied to the service provider and may pass data to/from the provider using an undocumented interface private to TSP++. This allows for the proxy to pass requests to the service provider which may need to gather information from the ACD or PBX before continuing with the request. This feature is not required, but is present if the same company is developing the provider and proxy or if the connection to the ACD/PBX software allows for only one client for both control and event notification.

The agent support is in a separate library and allows for the building of either NT services or NT/95 executables which perform the required TAPI agent proxy initialization and connection. The developer interacts with objects similar to the TSP++ objects but on the other side of TAPI (the application side).

Other Changes

Since TSP++ Version 3.0 is not backward compatible, some other changes have been made to ease the developers task of creating a service provider. These changes have been under discussion for some time but could not be implemented due to requirements of backward compatibility.

Request Object structures

In previous releases of the TSP++ library, a single object represented all the available asynchronous TAPI requests. That object had a method **GetDataPtr()** which allowed the retrieval of a structure containing request-specific information so that the provider could process the request.

This approach was taken so that the request object itself could be overridden by the developer to add additional information into it. As time went on however, very few users of TSP++ took advantage of this capability.

TSP++ Version 3.0 combines the request-specific data and the request object into a single object per request. For each asynchronous operation presented by TAPI, there is a request object derived from a common **CTSPIRequest** base which describes the operation completely.

For example, take the **lineSetupTransfer** request:

```
/*****************************************************************************/
//
// RTSetupTransfer
//
// This request object maps a lineSetupTransfer request.
//
/*****************************************************************************/
class RTSetupTransfer : public CTSPIRequest
{
// Class data
protected:
CTSPICallAppearance* m_pConsult;                    // Call appearance created as consultation
LPLINECALLPARAMS m_lpCallParams;                    // Call Parameters for consultation call

// Constructor
public:
        RTSetupTransfer(CTSPICallAppearance* pCall, DRV_REQUESTID dwRequestID
                CTSPICallAppearance* pConsult, LPLINECALLPARAMS lpCallParams);
        ~RTSetupTransfer();

// Access methods
public:
        CTSPICallAppearance* GetCallToTransfer() const;
        CTSPICallAppearance* GetConsultationCall() const;
        LPLINECALLPARAMS GetCallParameters();
};
```

When the request is processed by the service provider, it receives a basic **CTSPIRequest** object which has some known properties. One of the properties is the *type* of request, in this case, a **REQUEST_SETUPXFER**. The service provider can look at the type and know to cast the object to a **RTSetupTransfer** object instead –

```
RTSetupTransfer* pSetupTransfer = dynamic_cast<RTSetupTransfer*>(pRequest);
```

Once the object is cast, the service provider may then use the access methods of the object to get the pertinent information associated with the request.

This makes the implementation much simpler and minimizes the risk of not knowing what type of data is associated with which request.

In addition, the **REQUEST_xxx** values, which served a double purpose in the previous versions, are now only associated with asynchronous event objects which represent TAPI commands. This allows for a very hi-performance function pointer (vtable) style implementation of the processing functions within the service provider and is, in fact, the way the command processor is managed in our third-party TSP project currently being developed.

In the future, a wizard could be built to create the basic shell of the function processor and leaving “stub” functions for each supported event type.