Texas Instruments

The Avigo TOPS API

Version 1.10

Reference Guide

Table of Contents

Introduction 1

Overview of the Avigo Product

The Texas Instruments Avigo is an electronic organization solution that includes a handheld device and PC software. The package is designed to enable users to download information from the PC to the handheld device, providing easy access to that information when they are away from their PCs. The unit has the ability to accept or transfer information by wireless (infrared) or wired communication and can download and upload applications as well as data.

The Avigo includes the following built-in applications:

- Addresses
- Expenses
- Tasks
- Memos
- Schedules
- Sketch (a drawing pad)
- Calendar
- Calculator
- Database (defined by user)

Although the package includes a Personal Information Manager (PIM) program, the Application Program Interface (API) provides the ability for third-party software programs (such as scheduling software and name/address databases) to interface with the Avigo.

Internal Operations

The operating system for the Avigo handheld device is not an open operating system, but it does have the ability to download and upload applications.

The Avigo manages all data using tables. The Table-Oriented Protocol for Synchronization (TOPS) is a Texas Instruments protocol that defines the tables and objects used by the Avigo. The TOPS API is based upon the TOPS protocol.

If you are familiar with other Texas Instruments electronic organizers, you should be aware of the following changes in the Avigo system:

- The Avigo has the ability to create tables. In previous Texas Instruments organizers, the tables were a fixed set (addresses, reminders, notes).
- The user can define fields for the records in the tables as well as lists and categories for tables.
- Record and application IDs are assigned by the handheld, not by the PC.
- The system handles both wired and wireless communication.
- The handheld device can initiate a transfer of data (currently only when using wired connection).
- The handheld device can return an error message on any transfer.

Overview of the TOPS API

The TOPS API is a Windows 95 32-bit interface that provides a communication path from a PC to the Avigo. The TOPS API arranges the data from the PC data structures into the format needed by the Avigo without requiring the user to understand the Avigo formats.

The API is written in and uses run-time libraries of the C programming language. Calling applications conforming to the Windows standard calling convention should be able to use the API services, but header files are provided only for applications programmed in C.

The TOPS API:

- Provides a means of synchronizing the data in the PC software with the Avigo handheld device.
- Handles both wired and IrDA (Infrared Data Association) interfaces and can gracefully abort an interrupted transfer.
- Provides a method for creating new data, as well as deleting, modifying, and checking status of existing data stored on the Avigo.
- Has the ability to handle a request to synchronize anytime the Avigo initiates a transfer.
- Supports the Avigo's feature that permits users to define lists and categories.
- Supports the Avigo's ability to download applications.
- Provides a simple method of handling error messages from the Avigo.

Object Exchange Protocol

This API uses the IrDA Object Exchange Protocol (IrOBEX) for communication between the PC and the handheld device. The OBEX protocol is the second layer of transmission protocols, and overlays the IrDA protocol.

OBEX is a protocol for transmitting *objects*. An object may be a file or a smaller piece of information, such as a status code. OBEX consists of

- An object model, which includes an object and certain information about the object.
- A session protocol.

OBEX is a client-server protocol, and in this context, the PC software running the TOPS API is the *client*, and the device being communicated with (the Avigo handheld device) is the *server*.

Since the TOPS API handles all the protocol details, experience with the OBEX protocol is not required. If, however, you are interested in more detailed information about OBEX, consult the Infrared Data Association home page at ***http://www.irda.org***. (See the Standards section, OBEX documents.)

TOPS Objects

The Table-Oriented Protocol for Synchronization (TOPS) protocol specifies three different forms of objects recognized by the Avigo:

- ***Put object***: lets PC send new data to the handheld device.
- ***Get-Request object***: lets PC request data from the handheld device.
- ***Get-Response object***: lets handheld device return data to the PC.

Object Types on the Avigo

Each of the TOPS object forms listed above may consist of objects of various *types*. Objects of each object type exist as tables on the Avigo. The object types are listed as *typedef* statements in the tops_pda.h file (see *Appendix B: Header Files*).

The Avigo tables accessible through this API are:

- ***Table Info Table***
  (ID 0x8000) contains reference information about all the data tables on the Avigo. Each record within this table refers to a table on the Avigo. Each Avigo has a single *Table Info Table*.
- ***Application Table***
  (ID 0x8001) is a table for downloading and uploading applications. Information describing the application is embedded in the application, and is available through the Application Info Table.
- ***Application Info Table***
  (ID 0x8002) contains information about applications on the Avigo. Each record within this table refers to a specific application (which has the same record ID) on the handheld device.
- ***List Table***
  (ID 0x8003) is a table for managing lists. Each record corresponds to a list on the Avigo (such as the Type and Method lists used in Expenses).
- ***Category Label Table***
  (ID 0x8004) is a table for managing category labels, which are the selection items in drop-down lists (address types, for example).
- ***Application Preference Table***
  (ID 0x8005) is a table for managing user preferences. Each downloadable application is represented by a record in this table.
- ***Field Definition Table***
  (ID 0x8006) is a table for managing field attributes of data tables. Each data table has a corresponding record in the field definition table. For example, the Address table (a *Data Table*) has a corresponding record in the *Field Definition Table* that contains all the field definitions for a record in the Address list.
- ***Table Status Table***
  (ID 0x8007) is a table for maintaining the synchronization status for all Avigo data tables. Each table on the Avigo is represented by a record in this table.

- ***Dictionary Table***
  (ID 0x8008) contains only one record (Record ID 0), which is used to *Get* or *Put* (backup or restore) the user-specific dictionary entries for the T9 keyboard.

- ***Language Bundle Table***
  (ID 0x8009) is a table used to *Get* or *Put* the language-specific data, including the JustType User dictionary.

- ***Data Table***
  (ID range 0x0000...0x7fff) is a table for managing data (such as Addresses, Memos, Tasks, user databases, or data defined by downloadable applications).

Although a few of the Avigo tables have no relationship to the other tables, several of the tables function together to manage interrelated information.

Figure 1 shows the relationship between the data tables on the Avigo and the tables that manage information for them.

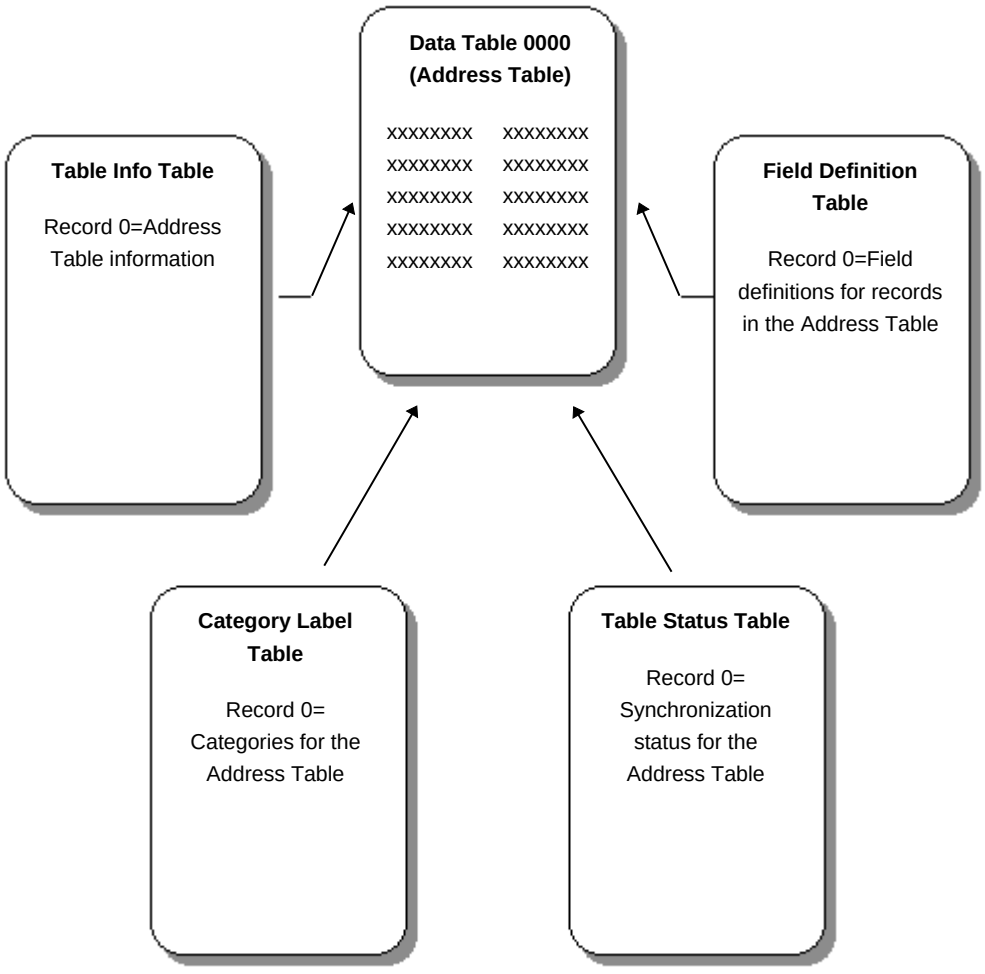

Figure 1: Avigo Tables

The Application tables (Application Info, Application, and Application Preference) also function together to manage information about the Avigo's applications.

Table IDs and Record IDs

The Avigo manages all data using tables. Each table consists of records; each record consists of fields.

The handheld device identifies tables and the records within the tables by their identifiers (Table IDs and Record IDs). These IDs are defined as constants in the tops_pda.h file.

Many of the function calls require both a *Table ID* and a *Record ID*. For a *Table Info Table*, the Record ID simply identifies the secondary table pointed to by the *Table Info Table*: Record 0 (0x0000) is the Address table, Record 1 is the Memo table, and so on. For the Address, Memo, Schedule, Task, Expense, Sketch, and other data tables, the Record ID refers to individual records within the tables. The Record IDs for the Address table, for example, refer to each address in the list, so that the first address is Record 0, the second is Record 1, and so on (see Figure 2). *Appendix A* of this document contains quick-reference tables that show the information fields associated with each defined table. The header file tops_pda.h (*Appendix B*) contains the structure for records for these defined tables.

Address Table ID 0x0000		
Record 0	Record 5	Record 10
Record 1	Record 6	Record 11
Record 2	Record 7	Record 12
Record 3	Record 8	Record 13
Record 4	Record 9	Record 14

Record 0 (0x0000)		
Synchronizer Key	Cellular	Home City
Category	Pager	Home State
Last Name	E-mail	Home Zip
First Name	Business Street	Home Country
Company	Business City	Birthday
Title	Business State	Notes
Business Phone	Business Zip	Index by
Fax	Business Country	Primary communication
Home Phone	Home Street	

Figure 2: Records in the Address Table

Users also have the option of defining new tables (databases) and the records and fields associated with the database.

The Table Type Parameter

Most of the TOPS API function calls include a TableType parameter. The Table Type refers to the table types discussed in the **Object Types** section (page 3), but, as used in the calls, consists of an identifier for the Table Type, which can be expressed either as the *Table ID* or the structure name. For example, if you are programming in C and have included our header file, and you need to get information from the *Table Info Table* on the handheld device, use 0x8000 or TOPS_TableInfoTable in the TableType parameter.

Notice that ten of the eleven table types listed in the previous section each have a single Table ID, indicating that each of these is a single table on the handheld device and requires only one identifier.

Table	Table ID	Constant
Table Info	0x8000	TOPS_TableInfoTable
Application	0x8001	TOPS_ApplicationRecordTable
Application Info	0x8002	TOPS_ApplicationInfoTable
List	0x8003	TOPS_ListTable
Category Label	0x8004	TOPS_CategoryLabelTable
Application Preference	0x8005	TOPS_ApplicationPreferenceTable
Field Definition	0x8006	TOPS_FieldDefTable
Table Status	0x8007	TOPS_TableStatusTable
Dictionary	0x8008	TOPS_DictionaryTable
Language Bundle	0x8009	TOPS_LanguageBundleTable

Figure 3: Table Identifiers

The *Data Table* type includes a range of identifiers since there can be multiple data tables. The predefined data tables are shown in Figure 4 below.

Data Table	Table ID	Constant
Address	0x0000	TOPS_AddressTable
Memo	0x0001	TOPS_MemoTable
Schedule	0x0002	TOPS_ScheduleTable
Task	0x0003	TOPS_TaskTable
Expense	0x0004	TOPS_ExpenseTable
Sketch	0x0005	TOPS_SketchTable

Figure 4: Data Table Identifiers

The remaining data Table IDs (0006 through 00ff) are either reserved for specific uses (such as user database tables: 8 through f), or reserved for assignment by the Avigo to new user-defined data tables or downloadable applications.

Communication Sequences

Each communication session with the handheld device must follow this pattern:

1. ***Detect communication initiated by the handheld device*. (Optional)**

   Function: CheckComm

   CheckComm is not required, but can be used to determine if the handheld has initiated a synchronization. If it has, then the PC does BeginSession.

2. ***Begin the communication session (includes defining the communication port and communication port type).***

   Function: BeginSession

3. ***Send or receive data.*** (See next section for a discussion of available functions.)

4. ***End the communication session (includes closing the open communication port).***

   Function: EndSession

Sending and Receiving Data

The following options exist for transferring data between the PC and the handheld device. Consult *Chapter 2: Function Calls* for detailed information about each call.

1. ***Create a record on the handheld.***

   GetSysInfo: PC requests new Record ID.

   PutRecord: PC sends the handheld a Data Record with the new Record ID.

2. ***Create a field on the handheld.***

   GetRecord: PC requests a particular Field Definition Record.

   PutRecord: PC sends the handheld an updated Field Definition Record.

3. ***Create a category on the handheld.***

   GetRecord: PC requests a particular Category Label Record.

   PutRecord: PC sends an updated Category Label Record.

4. ***Create a list on the handheld.***

   GetRecord: PC requests a particular list record.

   PutRecord: PC sends an updated List Record.

5. ***Create an application on the handheld.***

   GetSysInfo: PC requests a new Application ID.

   PutRecord: PC sends an Application.

6. ***Get the status of a particular record.***

   GetRecord: PC requests a particular record in a table. The status is returned along with the record.

7. ***Put data in a record in a table.***

   PutRecord: PC puts data for fields, applications, lists, categories, or preferences. May be called multiple times (if necessary) to send data to the handheld device.

   (In most cases, modified records should be transferred first, followed by deleted records, then by new records.)

8. ***Get data from a record in a table.***

   GetRecord: PC requests data from fields, applications, lists, or categories. May be called multiple times (if necessary) to obtain data from the handheld device.

9. ***Obtain options from handheld.***

   GetSysInfo: PC requests specific option information (settings) from the handheld.

10. ***Change options on the handheld.***

    PutSysInfo: PC sends option update to the handheld device.

Note

Not all personal options on the handheld device are read/write. Consult the tops_pda.h file (page 61) for a list of options that can be modified.

---

11. ***Terminate a procedure.***

    SetupCallback: Sets up a routine to be called by the TOPS API to terminate the current processing (in the event that a routine results in an error and stalls, or if the user cancels the process).

12. ***Detect communication initiated by the Avigo.***

    CheckComm: PC monitors for a send initiated by the handheld device.

Multiple Operations

The API provides several options for obtaining or setting the status of multiple items in one operation.

1. ***Get status for multiple records in one operation.***

   GetTags: PC requests status for the records of a table. A tag is simply a two-word data structure that contains a status and an ID that identifies the record.

2. ***Set status for multiple records in one operation.***

   PutTags: PC sets the status of records for a table.

Note

Since setting the delete tag in the status using PutTags is the method used to delete a record from a table, this routine can be used to delete multiple records from the handheld device in one operation.

It is recommended, however, that PC applications always delete records one at a time, since a timeout can occur if 15 (or more) records of a table containing 5000 records (or more) are deleted in a single tag table PUT. We anticipate that this problem will be corrected in Avigo versions 2.00 and greater.

Modifications to Data

A tag is simply a data structure that contains a status and an ID identifying the record. When you send a *Put* to the handheld device, you must include tag status bits that indicate whether the entry is being added, modified or deleted. The table (Figure 5) on the next page shows the status bits and command bits used for each of the table types discussed previously.

When status bits are sent to the handheld device by the PC, the handheld may need to perform an action or may need only to record the new status. The table below shows the result of setting the various status bits.

If…	**Then…**
Delete tag bit is set	Handheld device deletes both record body and record tag
Delete bit is set	Handheld device deletes record body
Other status bits set	Handheld device copies the received New, Modified, or Private status bits into the record's status

Note that you cannot delete a built-in table from the Avigo. You can, however, delete a downloadable application (by setting the Delete tag command bit for the Application Info Table) or a record within a table (by setting the Delete tag command bit for the data record).

Function Call Returns

All functions in this API return **1** for success, **0** for failure, or **2** for an unexpected response returned by the Avigo.

An *unexpected* response means that the Avigo indicated it was unable to perform the operation. If an unexpected response occurs, call GetUnexpectedResponse to get the error information from the Avigo.

Table Type	Record ID	Status/Command
Table Info (0x8000)	Table ID (0x0000 - 0x7fff)	**Tag Status Bits:** *1* Table Modified *4* Data Modified in Data table *5* Field Definition (of Table ID) Modified **Example:** 00000000 00100000 (Field definition modified)
Application Table (0x8001)	Application ID (0x0080 - 0x00ff)	**Tag Status Bits:** *0* New Application ID *2* Application Deleted **Tag Command Bits:** *3* Delete Application tag (deletes the application and its tag) **Example:** 00000000 00000000 (New Application)
Application Info Table (0x8002)	Application ID (0x0080 - 0x00ff)	All bits always 0
List Table (0x8003)	List ID (0x0000 - 0x000f)	**Tag Status Bit:** *1* List Modified
Category Label Table (0x8004)	Table ID (0x0000 - 0x0004)	**Tag Status Bit:** *1* Category Modified
Application Preferences Table (0x8005)	Application ID (0x0080 - 0x00ff)	**Tag Status Bit:** *1* Preference Modified
Data Table (Table ID: 0-0x7fff)	Record ID (0x0000 - 0xffff)	**Tag Status Bits:** *0* New Record ID *1* Record Modified *2* Record Deleted *8* Record Private **Tag Command Bit:** *3* Delete Record tag (deletes the record and its tag)
Field Definition Table (0x8006)	Table ID (0x0000 - 0x7fff)	**Tag Status Bits:** *1* Field Modified
Table Status Table (0x8007)	Table ID (0x0000 - 0x7fff)	All bits always 0
User Dictionary Table (0x8008)	Record ID (0x0000)	All bits always 0
Language Bundle Table (0x8009)	Record ID (0x0000)	All bits always 0

Figure 5: Status/Command Descriptions for GetTags and PutTags

Status Bits and the Data Record Lifecycle

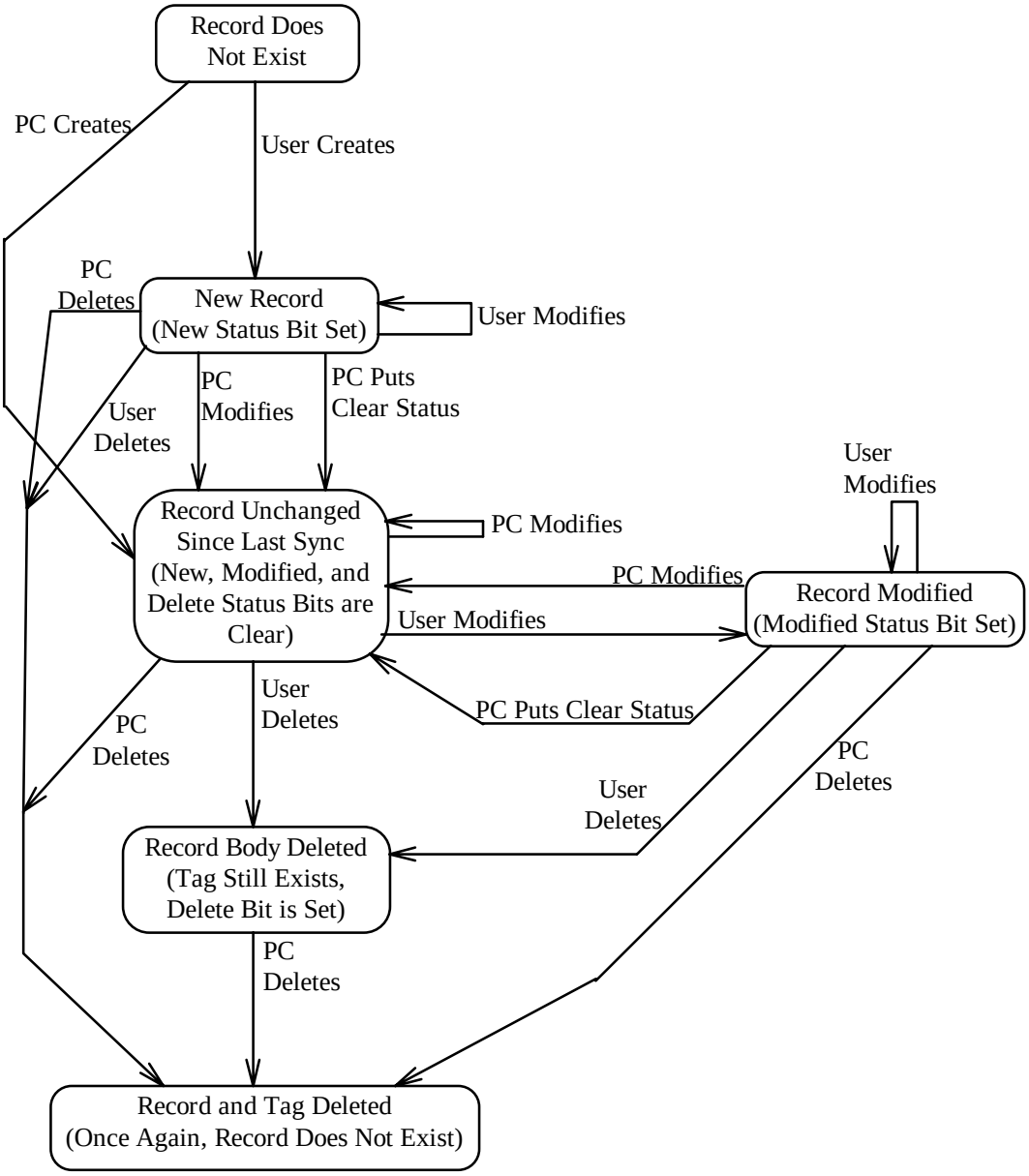

Figure 6: Data Record Lifecycle

PC Creates:	PC sends new record to the handheld device.
PC Modifies:	PC updates record on handheld. New, modified, delete, and delete tag bits are cleared in the status field of the Put Object used for the modification.
PC Deletes:	PC sends tag-table object with entry for the record having *delete tag* bit set.
PC Put Clear Status:	PC sends tag-table object with entry for the record having the new, modified, delete, and delete tag bits clear.
User Creates:	User creates record directly on handheld through user interface of handheld.
User Modifies:	User modifies record directly on handheld through user interface of handheld.
User Deletes:	User deletes record directly on handheld through user interface of handheld.

Status Bits in Put Objects and Put Tag Tables

For more information about status bits, see page 14 and PutRecord function, page 35.

Synchronization Guidelines

Connecting

The process of establishing a connection between the PC and the Avigo is different depending on whether the user has configured the system for wired communication (through the PC serial port) or for infrared communication.

Wired connection:

The two sides are unconnected until the user initiates communication from either the PC or the Avigo. If the user activates communication at your PC GUI (by clicking an on-screen button, for example), the TOPS API's BeginSession() call should be used. The PC polls for the device and, if Avigo is present and responds to the poll, establishes the connection. (The PC will poll twice during a call to BeginSession. We recommend that you call BeginSession() at least 3 to 5 times before giving up and alerting the user.) However, the user may be using the Avigo and tap on the Synchronize icon in the Synchronization screen. For this reason the PC needs to repeatedly call the TOPS API CheckComm() call. When CheckComm returns success, this means that the Avigo has signaled its presence and that the user wants to synchronize. The TOPS API BeginSession() call should then be used.

Infrared connection:

Because of the way the Microsoft IrDA stack for Windows 95 operates, communication can only be initiated by the user on the PC. For this reason, the Avigo hides the "Synchronize Now" button when the user selects infrared for connection type. When the user activates communication at the PC, call TOPS API's BeginSession() once : the call takes about 45 seconds to return (due to the Win95 IrDA stack) if the Avigo is not available to connect. If the Avigo is powered on and in range, it will connect to the Win95 IrDA stack.

Note: If an infrared dongle is of the newer type that acts just the same as a serial port, then the TOPS API can use it to communicate directly to the Avigo without the Microsoft IrDA stack. Use the TOPS API the same way as a wired connection described above, directing it to the com port for the dongle. The TOPS API uses its own internal IrDA stack for wired connections. This doesn't work with older non-standard dongles. For example, using the built-in IR port on a Texas Instruments TM5000 notebook will not work, although using a built-in IR port on the newer TM6000 will work.

Handheld ID

This System Information item (Type 2, subtype 2, item #0) is read first after a connection is established. It is used to determine which handheld the system is communicating with. If Avigo has just been reset, this is filled with 0's. When initializing this, the first byte is given a value identifying which PIM or Synchronizing System this handheld is being used with. This identifying number should be obtained from Texas Instruments. (Currently 0 is to not be used, TI uses 1, and PUMA uses 2). The remaining fifteen bytes should be randomly determined in such a way that a unique ID is obtained.

Passwords and the Unlock Command

If the password is set in the Avigo and the Avigo is in the Locked state, the only operation you can perform before unlocking the handheld with the Unlock Command (Type 3, #0) is getting the Handheld ID. If the password is set but the Avigo is not in the Locked state, you can do everything except get a record marked private, get a tag table of private records, and set the password.

If you think you know the password for the handheld (based on the handheld ID and a password the user may have previously provided for this purpose), send that in the unlock command. If Avigo complains that password was incorrect (with error 0 response code, (*Password Required*), try using an empty password field in unlock command (all bytes are 0) because the user may have performed the "reset forgotten password" operation on the handheld. If Avigo complains again, disconnect it and prompt user for the password, then reconnect and try the user-provided password. (You have to disconnect and then reconnect because the Avigo only allows two tries.) Any password tries after the second are rejected as incorrect until the next connection.

If you think the password has not been set, you do not need to use the unlock command. But if it has been set since the last synchronization, when you try to read a record marked private or get a tag table of private records, you will get an error 0 response code (*Password Required*) indicating the violation. It is best to just use the unlock command at the beginning (after connecting) with the password field empty (all 0 bytes). If you get an error 0 response code (*Password Required*), you then know to ask the user for the password.

Avigo passwords contain only upper case letters. The keyboard on the Avigo that is used to enter passwords does not allow entry of lower case letters.

Grouped Category A Sys Info Items

System information (later referred to as *Sys Inf*o items) consists of information and settings stored in the Avigo handheld, such as the Handheld ID, the database status, or the automatic optimization settings. Grouped Category A consists of all the Sys Info items that can be changed by the user of the handheld or by the handheld itself. This allows you to read all the user-/system-changeable data at one time. You cannot, however, write them back to the Avigo as a group. This is provided for the convenience of synchronizers that regularly examine this information or back it up. It is also much faster than reading each item individually.

Database Status

The Database Status Sys Info item (Type 2, subtype 0, item 0) is used to hold an identifying number that can be checked at the next synchronization. Usually just the time and date of the last synchronization is stored there for this purpose. If, during the next synchronization, you read this and it has a value different from the one you stored, you know that this handheld has been synchronized with some other PC or PIM and that you cannot rely on the New/Modify/Delete bits for synchronization. The Database Status is among the Grouped Category A Sys Info items.

Creating New Records

Before sending a new record, the PC must first request a new ID for that record from the Avigo. That ID is then used in the initial PUT of that record as well as for later updates. The Sys Info Item *New Record ID* is used for this purpose.

Status Bits, Tag Tables, and Synchronization

The PC changes an Avigo record's status bits by PUTting tag tables or by PUTting records.

When you PUT a tag table, the only Tags needed are of the records that have a changed status: the entire tag table is not required. Sending just a few tags in a tag table (to delete a record or clear its status) is most common.

If an Avigo record has been newly created then its *new* status bit is set. If an Avigo record is modified by the user then its *modified* status bit is set. In either case, after the change is handled by the PC, the status bits of the record are cleared by PUTting a tag table to the Avigo that includes a tag for the record. The tag's status bits are clear, except maybe the private bit, which is set if the record is private.

If an Avigo record has been deleted by the user, then its delete bit is set. Once the deletion is handled by the PC, the record's tag is no longer needed. The PC deletes the tag by PUTting the record's tag (in a tag table) to the Avigo with the delete-tag bit set. Note that if the user manually deletes a record that has its *new* bit set (that is., it has never been synchronized with the PC), then the record's tag is automatically deleted along with the record itself. The PC may delete a record body without deleting the tag by PUTting the record's tag (in a tag table) to the Avigo with the delete bit set. (Unlike a manual delete, PUTting a tag table with delete status bit for a record that has *new* status does not cause the tag to be deleted.)

When PUTting a record to the Avigo, there is no need to PUT a tag table to update the record's status bits because the status bits are included in the PUT. This may occur if a record was modified on both the PC and the Avigo and the user decided to keep the PC copy. In this case the record is PUT to the Avigo with the status bits clear.

Be certain that the changes a user has made to an Avigo record are stored on the PC before clearing status bits; otherwise, if communications is lost, synchronization cannot be easily guaranteed.

Tag Table of Table Info Records

This is the starting point for finding out what records in the system have changed since the last synchronization. At a high overview level, it covers all changes in the system not included in the Sys Info items. Get a tag table of table 0x8000 and all your steps in synchronizing follow—there's no need to examine each and every tag table in the system.

- If the Field Definition Modified bit is set for a data table, examine the Field Definition Record for that table.
- If the Data Modified bit is set for a data table, get a tag table of that data table's new, modified, and deleted records to use in updating the PC PIM with those changes.
- If the Modified bit is set for a data table, examine the Table Info Record for that table.
- In addition to tags for data tables, there are also tags representing the tables for Applications, Lists, Category Labels, and Downloadable Application Preferences : if the Data Modified bit is set for one of these tables, get a tag table of it to find out specifically which records in that table have changed.

Flash considerations

Flash is cheaper than SRAM, but there are some memory management costs. Flash is erased in 64K blocks. Erasing it changes all the bits to 1's. Once you write a bit to 0, it stays that way until you erase the whole 64k block. So, to modify a record you write the record somewhere else, in an unwritten part of flash, and mark the previous version of the record as deleted. To delete a record, you mark it as deleted. *Optimization* is the process of recovering those unused areas of flash marked as deleted. It involves copying the 64K to a RAM buffer, moving the undeleted data together to one end of the block, erasing the flash block, then copying from the RAM buffer back into the flash.

Flash is optimized automatically at regular intervals, usually nightly. Sometimes this is not enough, and you will be forced to perform an optimization in the middle of a synchronization. Unfortunately, an optimization takes a couple of minutes and so you want to minimize the chance of one occuring during synchronization.

It is hard to tell, before you begin, whether you have enough space in flash to do some lengthy operations without optimization. It is best to just modify and create records as needed until the Avigo complains that it has insufficient memory to complete an operation; then send a flash-optimize command to the Avigo. After optimization, then continue where you left off. The reason it is hard to determine whether there is sufficient space in flash is because of the blocked organization of the flash memory management system and the table management system used for data records.

Even if you could determine whether optimize is needed and perform it before the synchronization, there are still situations where you would have to optimize in the middle of a synchronization. For example, suppose the total size of several records about to be modified is greater than twice the size of the available space in flash (after an optimization). As you modify records, the prior copies become marked for deletion until you fill flash.You will have to perform at least two optimizations . A strategy to perhaps avoid the second optimization in this particular case, and to minimize the number of flash optimizations in general, is to perform a delete (not delete tag) of all the records you will soon be modifying in the current data table. If you get an "out of memory" response when modifying the records, then perform flash-optimize. Not only will the previous versions of those records already modified be cleaned out, but the soon-to-be modified copies as well.

It is better to do all modifications to a table's records together. Suppose you have been modifying records in table X and then modify a record in table Y: before the change to the record in table Y is begun, table X is saved in flash. Switching back and forth between tables and making changes to each of them, rapidly consumes flash.

Communication is dropped by the Avigo during an optimization. The PC has to reestablish communications afterward.

The "Wash"

The "Wash" is a term we use when potentially many records have to be modified because an attribute of a data table's field has been changed. A wash has to be performed whenever a category or custom field is deleted. A wash also has to be performed whenever a custom field's data type is changed. Because a wash is performed by modifying all of the records referring to a field attribute, it involves the previous discussion about flash considerations.

When a custom field is deleted from a table, all records having data in custom fields following the deleted custom field are included in the wash because the custom fields must be contiguous. For example, if custom field 3 is deleted, any data in custom fields 4 through 8 is moved to custom fields 3 through 7.

Avigo Character Set

The Avigo uses what is commonly referred to as the ANSI character set, which is actually an ISO standard. The characters from 0x80 to 0x9f, not yet defined by ISO, are those characters as displayed by TrueType fonts.

System Information

System information consists of information and settings stored in the Avigo handheld, such as the Handheld ID, the database status, or the automatic optimization settings. You need to be familiar with the various categories of Sys Info items to communicate successfully with the Avigo. The categories are described briefly below. For more detailed information, see Appendix A (page 41).

Category A Read/Write Objects

In addition to being read and written individually, all the Category A items may be read by the PC with a single GET of the Category A Grouped Object (Object Type 2, Subtype 7).

- Database status
- PC storage area for Category A
- Power-on message
- Automatic optimization settings
- Calendar application preferences
- Task application preferences
- Address application preferences

Category A Read Only

- Memory available
- Language bundle

Category B Read/Write

- Handheld ID
- PC storage area for Category B
- Synchronization screen message
- Splash screen image

Category C Read/Write

- Password

Category D

- New Record ID
- New Application ID

Installation Procedure

Before you attempt to use the TOPS API, you must do the following:

1. **Install the files.**

   The following files must be installed in the same directory, preferably your application directory.

   tops.dll
   The library for the TOPS API.

   tops.lib
   The import library for the tops.dll.

   io_error.lib
   The error library for the tops.dll; required to access the error buffer.

   io_tiobex.dll
   The library for TI OBEX communication.

   io_devices.h
   Contains constants for the various I/O devices.

   io_error.h
   Contains the definitions for the I/O API error buffer.

   tops_pda.h
   The header file for the TOPS API, which contains all typedefs, constants, and function definitions.

2. **Link the .dll to your application.**

   The TOPS API is a Windows 95 .dll called tops.dll. To gain access to the functions, the .lib file (tops.lib) must be linked into your application.

3. **Reference the header file in your source code.**

   Your source code should include all TOPS API header files listed in step 1.

4. **Place the .dll in your Windows 95 path.**

   Once your program is compiled and linked, ensure that these files are in the execution path:

   tops.dll (The TOPS library supplied with the TOPS API)
   cw3220.dll (The run-time library supplied with the TOPS API)

   kernel32.dll (Windows system library)
   user32.dll (Windows system library)
   wsock32.dll (Windows system library)

5. **Be sure that your computer is configured correctly.**

   If you are using the Microsoft Windows 95 IrDA stack for wireless communication, it must be properly installed and functioning. You can have the stack directed toward an IrDA port or a COM port. (The wired port usually uses COM1, and the IrDA port uses COM4.) You should be aware that you do not have to have the Windows 95 stack—the TI stack can handle both wired and wireless.

Note

When you are ready to compile, set the compiler data alignment option to BYTE. Do not use WORD or DOUBLE-WORD alignment.

Function Calls 2

Organization

This chapter includes descriptions of the Avigo TOPS API function calls. The functions are arranged alphabetically.

The following categories are included for each call:

Function	A brief description.
Syntax	Format for permitted parameters and arguments.
Parameters	Variables passed to and returned from the function call.
Description	A description of how the function works and additional information about the input required from the application programmer.
Example	Sample code using the function.

BeginSession

Function Starts a communication session with the handheld device.

Syntax

```
TOPS_BeginSession(int p_port,
                  IO_DeviceIdType p_device,
                  char *p_id_string);
```

Parameters

p_port

Logical COM port to use for communication to the handheld device if p_device is a COM device. This value is not the same as the actual physical COM port used for the IrDA connection. If p_device does not specify a COM device, then this parameter is ignored.

If p_device is IO_ComPort, p_port will have the actual port number used for communication, which is typically 1 for wired and 2 for infrared. If p_device is IO_IRDA, then the Windows 95 stack will be used and is the *virtual* COM port used by the Windows 95 stack, typically port 4.

p_device

The physical device layer for the connection, such as an IrDA device or a COM port. The valid values for this parameter are:

IO_ComPort (TI stack)
IO_IRDA (Windows 95 stack)

p_id_string

A pointer to a block of memory that will receive the Avigo identification character string returned by the handheld device. This string will be null terminated and contains version information that should be compared with the caller to ensure compatibility.

The buffer pointed to by this parameter must be large enough to contain the ID and the null terminator. The size of the buffer should be at least large enough to hold the longest ID string.

```
strlen("TI PPP PS/ LINK A 1.0/AVIGO V 1.01A ENG V 1.00")
```

This equals 46 including the null terminator. We recommend reserving extra memory (greater than 46 bytes) in case of changes.

Connect Messages A CONNECT message may include the Avigo identification character string (a WHO header) to indicate that the sender can support a custom subset or extension of OBEX. This is used only when there is a need to communicate custom objects that are specific to the device and not meant to be general interchangeable OBEX objects. If two connecting devices both include compatible WHO headers in their CONNECT messages, then these custom communications may be utilized.

The Avigo, in its response to a connect message, will send the ID string shown above. The Avigo refuses an OBEX connection if the received CONNECT packet does not have a WHO header that begins with the 18-characters “TI PPP PSCALC/LINK”. (Eighteen characters are used to prevent potential, though unlikely, conflicts with OBEX devices using 16-byte UUIDs for the WHO header.) To refuse a connection, the Avigo uses response code 0xE3 in its connect response instead of response code 0xA0 (OK).

Description

This call initializes the API and defines the COM port and port type that will be used for this session. This function must be the first function called. If it is not called, or if it returns a FALSE value (**0**), all subsequent calls to the API will return FALSE.

After this function returns **1** (TRUE), indicating a successful connection, you must issue another call to receive data from or transfer data to the handheld device.

Note

There is nothing that prevents the user from using the TI stack for communication over the infrared port rather than the serial port, (or for using the Windows 95 stack for the serial port rather than the infrared port). The original intent of the TI stack, however, was to provide a stand-alone stack for the wired port so that the Windows 95 stack and the TI stack could function simultaneously on different ports.
The user can call this routine and switch back and forth on the input parameters if it is not known which device or port will be used for the connection.

Example 1

```
// Begin a communication session with the handheld using the IrDA
// port and the Win95 stack

  Port = 4;
  Device = IO_IRDA;
  ReturnStatus = TOPS_BeginSession(
    Port,
    Device,
    IdString
    );

  if (ReturnStatus == TRUE) {
    //Avigo identification string is in IdString
    //...continue normal processing...
  }
  else {
    //...Error processing...
  }
```

Example 2

```
// Begin a communication session with the handheld using the COM
// port and the TI stack

  Port = 1;
  Device = IO_ComPort;
  ReturnStatus = TOPS_BeginSession(
    Port,
    Device,
    IdString
    );

  if (ReturnStatus == TRUE) {
    //Avigo identification string is in IdString
    //...continue normal processing...
  }
  else {
    //...Error processing...
  }
```

CancelCallback

Function Removes a callback that was set up using SetupCallback.

Syntax TOPS_CancelCallback(void);

Parameters None.

Description This function call removes a callback that was set up using the SetupCallback function.

Example

```
ReturnStatus = TOPS_CancelCallback();
//...Continue with Puts, Gets, etc. here
//...Any previously registered callback function is no longer called
```

CancelDisconnectWarningCallback

Function Removes a callback that was set up using SetupDisconnectWarningCallback.

Syntax TOPS_CancelDisconnectWarningCallback(void);

Parameters None.

Description For more information, see SetupDisconnectWarningCallback, page 40.

Example

```
ReturnStatus = TOPS_CancelDisconnectWarningCallback();

//...Continue with Puts, Gets, etc. here
//...Any previously registered callback function is no longer called
```

CheckComm

Function Checks to see if Avigo is requesting communication.

Syntax

```
IO_DeviceIdType TOPS_CheckComm(BYTE p_port,
                               IO_DeviceIdType p_device);
```

Parameters

p_port

Tells the software which logical COM port to use to listen for communication with the handheld device. If the TI stack is to be used (for wired communication), this value is COM1 or COM2. At this time, the Avigo handheld device cannot request connection by infrared.

p_device

Identifies the type of connection to check. At the present time, the device is always IO_ComPort.

Description This function call monitors for a send initiated by the handheld device. Since the handheld has the ability to start a transfer whenever the connection is wired, this call is necessary to enable the PC to receive the request.

Example

```
//  Check the ability to communicate with the handheld

CheckCommPort = 1;
CheckCommDevice = IO_ComPort;
CheckCommIdType = TOPS_CheckComm(CheckCommPort, CheckCommDevice);
switch(CheckCommIdType)
{
case InvalidLowDevice : //...Error processing...
case ...
default               : //...Normal processing...
}
```

EndSession

Function Ends a communication session with the handheld device.

Syntax TOPS_EndSession(void);

Parameters None.

Description When this function is called, the PC initiates an OBEX disconnect sequence, closes the COM port on the PC, and closes the TOPS API. A disconnect is sent to the handheld device.

This must be the last API function called. At this time, this function always returns TRUE and ends the session.

Example

```
//  End the communication session with the Avigo

ReturnStatus = TOPS_EndSession();
```

GetRecord

Function Returns a data record from a table on the Avigo.

Syntax

```
TOPS_GetRecord(TOPS_TableType p_table_type,
               TOPS_RecordId p_record_id,
               WORD *p_returned_rec_status_ptr,
               TOPS_Record *p_returned_rec_ptr);
```

Parameters

p_table_type

An identifier for the table that contains the record you need. (Refer to page 28 and Appendix B for a list of the table types.)

p_record_id

The numeric identifier (Record ID) for the record being requested from the handheld device.

p_returned_rec_status_ptr

A pointer to a memory location that will receive the status of the returned record.

p_returned_rec_ptr

A pointer to a memory location that will receive the returned record.

Description This function call retrieves a data record from the handheld device. Specify the record needed by entering the appropriate input parameters.

Example

```
//  Get an address record and its modification status from the PC
//  Companion

TableType = TOPS_AddressTable;
RecordId = 0xa1b2c3d4;
ReturnStatus = TOPS_GetRecord(
  TableType,
  RecordId,
  &ReturnedRecStatus,
  (TOPS_Record *)&AddressRecord
  );

if (ReturnStatus == TRUE) {
  //Record modification status is in ReturnedRecStatus
  //AddressRecord contains valid record data
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

GetSysInfo

Function Gets specified system information from the handheld device.

Syntax

```
TOPS_GetSysInfo(TOPS_SysInfoItems p_item_type,
                TOPS_SysInfo Parms *p_parameters,
                TOPS_SysInfo *p_item_ptr);
```

Parameters

p_item_type

An identifier for the item type to be retrieved from the handheld device. See the header file tops_pda.h, *Sys Info* section**,** for a list of specific identifiers for the system information.

p_parameters

The parameter for the *get*. It is needed only if p_item_type is NewRecordItem or NewApplicationIdItem.

p_item_ptr

A pointer to a memory location that will receive the specified system information.

Description This function call retrieves system information from the handheld device. The Avigo contains a large number of data items that are classified as system information, including the handheld device ID, time settings, alarm settings, personal preferences, and so on. Some items are *read-only*; others are *read/write*. For a complete listing of these items, see tops_pda.h, *SysInfo* section (page 81).

Example

```
//  Get the database status from the Avigo

SysInfoType = TOPS_DatabaseStatusItem;
SysInfoParms.AppIdParm.TOPS_NewAppSize = 0;
  //Used with D class items only
ReturnStatus = TOPS_GetSysInfo(
  SysInfoType,
  &SysInfoParms,
  (TOPS_SysInfo *)&DatabaseStatus
  );

if (ReturnStatus == TRUE) {
  //The database status is in DatabaseStatus
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

GetTags

Function Gets tags of all the records of a table.

Syntax

```
TOPS_GetTags(TOPS_TableType p_table_type,
             TOPS_Selector p_selector,
             unsigned int *p_num_of_returned_tags,
             TOPS_TagList *p_returned_tags_ptr);
```

Parameters p_table_type

An identifier indicating which table to get tags for. The Table Types are:

- Table Info Table (0x8000) TOPS_TableInfoTable
  Since this table contains status for all other tables, requesting tags for this table is the quickest way to determine which tables have changed in the handheld device.
- Application Table(0x8001) TOPS_ApplicationRecordTable
- Application Info Table (0x8002) TOPS_ApplicationInfoTable
  This table returns all **0**s, since no status bits are defined for App Info Table.
- List Table (0x8003) TOPS_ListTable
- Category Label Table (0x8004) TOPS_CategoryLabelTable
- Application Preference Table (0x8005) TOPS_ApplicationPreferenceTable
- Field Definition Table (0x8006) TOPS_FieldDefTable
- Table Status Table (0x8007) TOPS_TableStatusTable
  This table returns all **0**s, since no status bits are defined for Table Status Table.
- User Dictionary Table (0x8008) TOPS_DictionaryTable
  This table returns all **0**s, since no status bits are defined for the Dictionary Table.
- Language Bundle Table (0x8009) TOPS_LanguageBundleTable
- Data Table (0..0x7FFF)
  See page 5 and Appendix A for the various data tables.

p_selector

Selects the tags that you want to obtain. The selector can be

All Returns ALL tags of the table.

New Returns only tags of records that have the New bit set.

Modified Returns only tags of records that have the Modified bit set.

Deleted Returns only tags of records that have the Deleted bit set.

Private Returns only tags of records that have the Private bit set.

You can also select any combination of the above options.

p_num_of_returned_tags

A pointer to a location in memory that will receive the number of tags that are returned by the handheld device.

p_returned_tags_ptr

A pointer to a buffer that will receive the returned tags.

Description

This function gets tags for the records in a table. A tag is a two-word data structure that contains the Record ID and the status of the record. The status tells if the record is new, modified, or deleted. In some cases, it also indicates whether or not it is an exclusive access (private) record.

Note

If you need status for a single record, use GetRecord. The API returns the status with the record.

Example

```
//  Get the tags for new Avigo address records
TableType = TOPS_AddressTable;
TagsSelector = TOPS_GetNew;
ReturnStatus = TOPS_GetTags(
  TableType,
  TagsSelector,
  &NumTags,
  &TagList
  );

if (ReturnStatus == TRUE) {
  //The number of tags returned is in NumTags
  //The first tag is pointed to by TagList
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

GetUnexpectedResponse

Function Returns the unexpected response object (the error message) to the caller.

Syntax

```
TOPS_GetUnexpectedResponse(TOPS_ErrorCategory *p_error_category_ptr,
                           TOPS_ErrorId *p_error_id_ptr);
```

Parameters

p_error_category_ptr

A pointer to the memory location that will receive the error category from the handheld device.

p_error_id_ptr

A pointer to the memory location that will receive the error ID from the handheld device.

Description This function call retrieves error information from the handheld device. The *error category* may be **0** (CommonError) or **1** (SpecificError). The *error ID* for common errors are:

0 Password required
1 Requested object not found (No record with that ID)
2 Insufficient space in memory or flash to complete operation
3 Unimplemented OBEX Opcode
4 ID unavailable because ID table is full
5 Bad usage
6 Object was incorrect size (usually a fixed-length Sys Info object)
7 Low battery. Unable to complete operation.

Error 5 (Bad usage) indicates that one of the following occurred:

- PUT of a read-only object.
- Unrecognized type or subtype used.
- Unknown Table ID used.
- Out-of-range Record ID used.
- Syntax error in record

Note

This version of the API does not include Specific Errors.

Currently, this function always returns successfully (1).

Example

```
//  A call to TOPS_GetRecord returns an unexpected response

TableType = (TOPS_TableType)1050; //Some data table
RecordId = 0xa1b2c3d4;            //Some record in that table
ReturnStatus = TOPS_GetRecord(
  TableType,
  RecordId,
  &ReturnedRecStatus,
  (TOPS_Record *)&TableRecord
  );

if (ReturnStatus == TOPS_UnexpectedResponse) {
  //Avigo returned an unexpected response

  ReturnStatus = TOPS_GetUnexpectedResponse(&ErrorCategory,
&ErrorId);
  switch(ErrorCategory)
  {
    case CommonError:
      switch(ErrorId)
      {
        case Password_Required  : //...Error processing...
        case ...
      }
      break;
    case RequestSpecificError:
      //...Error processing...
      break;
    ...
  }
}
```

PutCommand

Function Changes system information on the handheld device.

Syntax

```
TOPS_PutCommand(TOPS_CommandType p_item_type,
                TOPS_CommandParameter *p_item_ptr );
```

Parameters

p_item_type

An identifier for the type of item being sent to the handheld device, such as TOPS_Reset.

*p_item_ptr

A pointer to the item being sent to the handheld device.

Description PutCommand allows you to change system settings on the Avigo. Use of this command may be necessary if you need to compact the memory on the organizer or prompt the user for a password to unlock the organizer.

You can use the following item types with the Put command:

0	TOPS_UnlockWithPassword
1	TOPS_CompactMemory
2	TOPS_SyncMessages
3	TOPS_DeleteUserDictionary

UnlockWithPassword. 8 bytes. For passwords shorter than 8 bytes, the byte following the password is 0x00 (any remaining bytes after the 0x00 may have any value). If the string does fill the entire field, then there will be no 0x00 byte. An unset password has 0x00 as the first byte.

If password is not set, then this command is not required. Also, if password is not set and the unlock command is sent to HH with an empty password field (first byte is 0x00), the response must be OK (0xA0).

If the submitted password in the unlock command is incorrect, error response 0, *password required*, is returned to the PC.

Additionally, if password is not set (empty) and a nonempty password is submitted in the unlock command, then the response must also be error response code 0, “password required”.

This command may be attempted twice. Any further attempts will always be responded to with error response 0 until the next future connection, even if the submitted password is correct.

(***Note*** : the Avigo 10 does not follow this rule, however it would still take an average of 42 months to break into an Avigo 10 with a 6 character password).

Compact/OptimizeMemory. When this command is sent to the handheld:

1. The handheld returns OK.
2. Sends RD (request disconnect) at the IrLAP layer.
3. Drops connection.
4. Performs Compact/Optimize of memory.
5. Returns to synchronize screen.

The connection is dropped while optimization takes place because Avigo does not have enough RAM to maintain a connection while performing compact/optimize. The Avigo returns to the Synchronize Screen when done in order to allow the PC to re-establish a connection. This is necessary for PC to be able to handle memory full conditions that can be resolved by reclaiming the space held by deleted records.

SyncMessages. There are 3 fields. Each is a maximum 64-bytes-long message. (If message is shorter than 64 bytes, the first byte following the message is 0x00.)

1. Message to display during synchronization. This is displayed immediately when received.
2. Message to display at end of synchronization. This is displayed in a message box when TOPS disconnect is received or if communications is lost.
3. Message for synchronization results screen. This is displayed after user exits message box and whenever user returns to synchronization screen. (This last message is stored in System Info Category B Read/Write (Type 2, Subtype 2) #3 when this command is received.)

This command should be sent at least twice during a Synchronization session: once at the beginning and once at the end. When sending this command at the beginning, field 1 should have text telling the user that synchronization is proceeding, and fields 2 and 3 should have text that would be meaningful if communications was disrupted and lost. When sending this command at the end, field 1 can be empty, field 2 could say *Synchronization Completed*, and field 3 could be *Last Synch on 4/3/97 2:42 PM*.

Carriage returns may be used in message to position cursor at the beginning of the next line.

See the table on the next page for information about how the display works during synchronization.

1. If the first byte of message #1 is 0x0, use the previously received message #1. (If message command has not been received before, use default.)
2. If the first byte of message #2 is 0x0, use the previously received message #2. (If message command has not been received before, use default.)
3. If the first byte of message #3 is 0x0, do not change message #3.

These last three rules are useful for two reasons:

1. Some communications should not change message #3. For example, a program that examines handheld data but does not change it.

2. Many times, the default error messages would be useful.

In the handheld, these 0-field rules can be implemented by just looking at the first byte in each field in the received packet before updating messages in internal RAM.

DeleteUserDictionary. No parameters. This commands Avigo 10 to erase the bank of flash containing the T9 User Dictionary. Avigo will disconnect, erase the bank, then allow the PC to reconnect. Takes under 10 seconds. This must be used before PUTting a T9 user dictionary object (Type 0, Subtype 9, table 0x8008).

Example

```
//Example call to TOPS_PutCommand

//  Command the Avigo to compact its memory

//...Initialize CommandParameter here if needed for the
//   particular Sys Info type ...
CommandType = TOPS_CompactMemory;
ReturnStatus = TOPS_PutCommand(
  CommandType,
  &CommandParameter
  );

if (ReturnStatus == TRUE) {
  //Command was successfully sent to the Avigo
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

Event	Action if message command **was not** received from PC.	Action if message command **was** received from PC.
1. User taps *Synchronize Now*, or PC starts communication.	Display *Synchronizing*.	(Cannot happen)
2. Message command received from PC.	Display message #1 immediately.	Display message #1 immediately.
3. Communications with PC is lost.	Display default error message until user taps *OK* button.	Display message #2 until user taps *OK* button.
4. PC sends TOPS/OBEX disconnect.	Display *Synchronization Complete* for 3 seconds.	Display message #2 for 3 seconds.
5. After event 3 or 4 above, when message box disappears.	Show message #3. (In this case, message was received during previous communication. It has been stored in Sys Info Category B Read/Write item #3)	Show message #3.
6. Whenever user goes to Synch Screen later.	Show message #3.	Show message #3.

Figure 7: How the Avigo Display Works During Synchronization

PutRecord

Function Puts a data record on the handheld device.

Syntax

```
TOPS_PutRecord(TOPS_TableType p_table_type,
               TOPS_RecordId p_record_id,
               WORD p_record_status,
               TOPS_Record *p_record_ptr);
```

Parameters

p_table_type

An identifier for the table to which the record is to be added. This can be any of the valid table types. (Refer to page 28 for a list of the table types.)

p_record_id

The numeric identifier for the record being sent to the handheld device.

p_record_status

The status of the record being sent to the handheld device. Valid statuses are:

0	New
1	Modified
2	Deleted
8	Private

p_record_ptr

A pointer to the record being sent to the handheld device.

Description This function call sends a data record to the handheld device. When status bits are sent to the handheld by the PC, either through a Put of a record or Put of a Tag Table, the handheld may need to perform certain actions or may only need to record the newly provided status. Here is the rule for status received from the PC:

```
if (delete tag bit set)
{
    Delete both record body and record tag.
}
else
{
    if (delete bit set)
    {
       Delete record body.
    }
    Copy the received New, Modified, Delete, and Private status bits
into the record's status.
    (Note: only one of the New, Modified, and Delete bits may be set. )
}
```

Example

An example where this rule is used:

When a friend transfers some addresses from his or her PC to your handheld, the PC sends the records down with the New status bit set. When you later synchronize with your home PC, the New status bit for that record alerts the PC to the existence of the new record. The effect is the same as if your friend had entered the new addresses into your Avigo by hand.

A special case to be aware of :

Consider a record that has been deleted by user on Avigo. The body is deleted, but the tag remains and the delete bit is set. There are some situations where the PC will want to restore the record at the next synchronization. In this case, the Avigo should accept the record and store it under the given ID. The PC will set the status in the Put Object so that the delete bit is clear. Essentially, the record tag has been given a new body to associate with, and the delete bit is now clear.

Some related finer points :

If a data record in the Avigo has its delete status bit set, and you do a PUT record to that ID, the record is accepted and status bits are taken from the status in the PUT.

If you do GET-request a record that has delete status bit set, the Avigo responds with error code 1 : *Requested object not found.*

Example

```
//  Add a Table Info record and its status to the Avigo

//...Initialize the TableInfoRecord here...
//...
TableType = TOPS_TableInfoTable;
RecordId = 0xa1b2c3d4;
RecStatus = TOPS_RecordNew;
ReturnStatus = TOPS_PutRecord(
  TableType,
  RecordId,
  RecStatus,
  (TOPS_Record *)&TableInfoRecord
  );

if (ReturnStatus == TRUE) {
  //Record was successfully added to the Avigo
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

PutSysInfo

Function Puts system information items on the handheld device.

Syntax

```
TOPS_PutSysInfo(TOPS_SysInfoItems p_item_type,
                TOPS_ReadWriteSysInfo *p_item_ptr);
```

Parameters

p_item_type

An identifier for the type of item being sent to the handheld device, such as TOPS_DatabaseStatusItem (Category A Read/Write, Item 0) or TOPS_DailyAlarmItem (Category A Read/Write, Item4). For a complete list of the system information Read/Write items, see tops_pda.h, *SysInfo* section (page).

p_item_ptr

A pointer to the item being sent to the handheld device.

Description This function sends system information to the handheld device. The Avigo contains a large number of data items that are classified as system information, including the handheld device ID, time settings, alarm settings, personal preferences, and so on. Some items are *read-only*; others are *read/write*. You can use the PutSysInfo call only for read/write items. For a complete listing of these items, see tops_pda.h, *SysInfo* section (page 81).

Example

```
//  Put information in the Avigo storage A area

//...Initialize the PC_Storage_A array here...
//...
SysInfoType = TOPS_PC_Storage_A_Item;
ReturnStatus = TOPS_PutSysInfo(
  SysInfoType,
  (TOPS_ReadWriteSysInfo *)&PC_Storage_A
  );

if (ReturnStatus == TRUE) {
  //SysInfo item was successfully added to the Avigo
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

PutTags

Function Sets the status of a record or records in a table.

Syntax

```
TOPS_PutTags(TOPS_TableType p_table_type,
             WORD p_num_tags,
             TOPS_TagList *p_tags_ptr);
```

Parameters

p_table_type

An identifier for the table type for which you are changing record tags.

p_num_tags

Number of tags to change. This number must be equal to the number of tags pointed to by p_tags_ptr.

p_tags_ptr

A pointer to a memory location that will receive the list of new tags for the specified records and table. The list will contain one tag for each record to be modified.

Description This function sets the status for records in a table. A tag is a two-word data structure that contains a status and an ID that identifies the record (the Record ID). The status indicates whether the record is new, modified, or deleted. (See p_selector parameter on page 28 for more details on the status.)

Example

```
//  Add a tag to the Avigo for a memo record

//...Initialize the TagList here...
//...Initialize the number of tags NumTags here...
//...
TableType = TOPS_MemoTable;
ReturnStatus = TOPS_PutTags(
  TableType,
  TagsNumTags,
  &TagList
  );

if (ReturnStatus == TRUE) {
  //Tag was successfully added to the Avigo
  //...continue normal processing...
}
else {
  //...Error processing...
}
```

SetupCallback

Function Sets up a routine to be called to terminate the current processing.

Syntax TOPS_SetupCallback(int (_USERENTRY *p_IO_Callbackfunction)(void));

Parameters p_IO_Callbackfunction

A pointer to a third-party function that will be called if the current process needs to be terminated.

Description This function call can terminate processing if a routine stalls or if the user cancels a process.

Example

```
//  Have the API call MyCallBackFunction during long operations.

//  MyCallBackFunction terminates the API operation if the user has
//  requested it be halted (via some mechanism not shown here).

int _USERENTRY MyCallBackFunction (void) {
  if (UserHasRequestedAbort) {
    return FALSE; //Stop the API operation
  }
  else {
    return TRUE; //Continue the API operation
  }
}


//...processing...
ReturnStatus = TOPS_SetupCallback(MyCallBackFunction);
//...Continue with Puts, Gets, etc. here...
```

SetupDisconnectWarningCallback

Function Sets up a routine that will be called if IrLAP (Infrared Link Access Protocol) does not receive a response from the handheld device.

Syntax TOPS_SetupDisconnectWarningCallback(void (_USERENTRY *p_IO_Callbackfunction)(int));

Parameters p_IO_Callbackfunction

A pointer to the callback function.

Description Use this function to set up a routine that will be called by the API when the IrLAP layer of the TI stack has not received a response from the handheld. This could be caused by the handheld being removed from the cradle, the cable being removed, or some other occurrence that prevents the handheld device from responding. The API calls the callback with a parameter indicating the current disconnect count. (IrLAP sends the first Warning when the disconnect count reaches 3, so the first number received will be 3.) If the handheld starts responding, the callback will be called with a value of 0xffff. If it never returns, it will get the value 0xfffe.

This function is intended to allow the API user to display a message to the end user when an obstruction in the data path occurs. The API user knows that the message can be removed when it receives the 0xffff value. This is not as likely for wired communication as in a true infrared usage.

Example

```
//  Have the API call MyDisconnectWarningCallBackFunction when the
//  communication link is broken. MyDisconnectCallBackFunction
//  displays a message window while the link is broken.

void _USERENTRY MyDisconnectWarningCallBackFunction (int ErrorCount)
{
  if ((ErrorCount >= 3) && (ErrorCount < 0xFFFE)) {
    //...display message window notifying user of communication
break
  }
  else if (ErrorCount == 0xFFFE) {
    //...remove the message window
    //...error processing
  else if (ErrorCount == 0xFFFF) {
    //...remove the message window
    //...continue normal processing
  }
}



//...processing...
ReturnStatus =
TOPS_SetupCallback(MyDisconnectWarningCallBackFunction);
//...Continue with Puts, Gets, etc. here...
```

Appendix A: Table and Field Reference

Table IDs

ID	Table
0x0000	Address table
0x0001	Memo table
0x0002	Schedule table
0x0003	Task table
0x0004	Expense table
0x0005	Sketch table
0x0008 to 0x000f	User-defined tables
0x0080 to 0x00ff	Downloadable applications
0x8000	Table Info table
0x8001	Application table
0x8002	Application Info table
0x8003	List table
0x8004	Category Label table
0x8005	Application Preferences table
0x8006	Field Definition table
0x8007	Table Status table
0x8008	T9 Keyboard User Dictionary
0x8009	Language Bundle

Address Record Fields

For structure, see page 66. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Category	Category	Yes	1
2	Last Name	Text128	Yes	128
3	First Name	Text128	Yes	128
4	Company	Text128	Yes	128
5	Title	Text128	Yes	128
6	Business Phone	Text128	Yes	128
7	Fax	Text128	Yes	128
8	Home Phone	Text128	Yes	128
9	Cellular	Text128	Yes	128
10	Pager	Text128	Yes	128
11	E-mail	Text128	Yes	128
12	Business Street	Text128	Yes	128
13	Business City	Text128	Yes	128
14	Business State	Text128	Yes	128
15	Business Zip	Text128	Yes	128
16	Business Country	Text128	Yes	128
17	Home Street	Text128	Yes	128
18	Home City	Text128	Yes	128
19	Home State	Text128	Yes	128
20	Home Zip	Text128	Yes	128
21	Home Country	Text128	Yes	128
22	Birthday	Date	Yes	3
23	Notes	Text512	Yes	512
24	Index by *	Integer	Yes	2
25	Primary communication **	Integer	Yes	2
26-33	Unused	Unassigned	No	128

* Index by: 0=Last name/First name
1=Company name/Last name

** Primary communication: 0=Office (phone 1), 1=Fax, 2=Home, 3=Cellular, 4=Pager, 5=email

When you save an address record, the Avigo checks the name fields :

```
If Last name is empty  && First name is empty
      then
         index by = company name (1)
      else
         index by = last name, first name (0)
```

Memo Record Fields

For structure, see page 67. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Category	Category	Yes	1
2	Title	Text256	Yes	256
3	Memo	Text3.5K	Yes	3584
4-7	Unused	Unassigned	No	128

Schedule Record Fields

For structure, see page 67. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Category	Category	Yes	1
2	Date	Date	Yes	3
3	Time	ScheduleTime	Yes	6
4	Description	Text256	Yes	256
5	Repeat	Repeat	Yes	9
6	Alarm	Alarm	Yes	1
7	Note	Text512	Yes	512
8-15	Unused	Unassigned	No	128

Repeating Schedule Items:

A repeating event is defined by the repeat criteria (the repeat field in the data record) that the user enters. If an event is repeating type, the date field is set to 0x00. This differentiates the date field from one that has a date or no date (filled with 0xff's). If an event is not repeating, the repeat field will usually be set to 0xff, but may contain repeat criteria if the user sets the repeat entry screen of a record and then decides not to make it a repeat.

Task Record Fields

For structure, see page 67. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Category	Category	Yes	1
2	Start Date	TaskDate	Yes	300
3	Due Date	TaskDate	Yes	300
4	Complete Date	CompleteDate	Yes	300
5	Description	Text256	Yes	256
6	Priority	Priority	Yes	1
7	Repeat	Repeat	Yes	9
8	Note	Text512	Yes	512
9-16	Unused	Unassigned	No	128

Repeating tasks are not supported by the Avigo 10. PUTs of Task records should have the repeat field filled with 0xff's.

Expense Record Fields

For structure, see page 68. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Category	Category	Yes	1
2	Date	Date	Yes	3
3	Amount	Currency	Yes	8
4	Description	Text128	Yes	128
5	Type	ListType1	Yes	16
6	Merchant	Text128	Yes	128
7	Location	Text128	Yes	128
8	Method	ListType2	Yes	16
9	Note	Text512	Yes	512
10-17	Unused	Unassigned	No	128

Sketch Record Fields

For structure, see page 68. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Sketch Type	Integer	Yes	2
2	Title	Text16	Yes	16
3	Drawing	Drawing	Yes	3600

The Sketch Type (field 1) is reseerved for future use. For the present, sketch records should always be PUT with 0 in this field. If you GET a record and it has Sketch Type field not zero, do not interpret that sketch. Tell the user it is of an unknown type.

User-defined Table Record Fields

For structure, see page 68. For explanation of data types and fixed type, see page 50.

	Field name	Data type	Fixed type?	Maximum length
0	Synchronizer Key	Synchronizer Key	Yes	128
1	Unused 1	Unassigned	No	128
2	Unused 2	Unassigned	No	128
3	Unused 3	Unassigned	No	128
4	Unused 4	Unassigned	No	128
5	Unused 5	Unassigned	No	128
6	Unused 6	Unassigned	No	128
7	Unused 7	Unassigned	No	128
8	Unused 8	Unassigned	No	128
9	Unused 9	Unassigned	No	128
10	Unused 10	Unassigned	No	128
11	Unused 11	Unassigned	No	128
12	Unused 12	Unassigned	No	128
13	Unused 13	Unassigned	No	128
14	Unused 14	Unassigned	No	128
15	Unused 15	Unassigned	No	128
16-30	"	"	No	128
31	Unused 31	Unassigned	No	128
32	Unused 32	Unassigned	No	128

Unlike data tables for standard applications, user data tables can be created and deleted. Specific procedures must be followed when creating and deleting user data tables.

To create:

1. Select table info record with empty name field.
2. PUT field definition record for the selected table.
3. PUT table info record for the selected table. Name field must be non-empty.
4. PUT data records to the selected table.

To delete:

1. Delete all data records from the table.
2. PUT table info record, for that table, with empty name field.

The presence or absence of a name in the name field acts like a switch to show or hide the user data table. When the user wants to create a new user data table, the Avigo selects a user data table with an empty name in its table info record. (The Avigo ignores the existing field definition record for that table and begins with one where field data types are unassigned. That is why you do not have to PUT an empty field definition record in the procedure listed above for deleting user data tables.)

After reset, there are no user data tables. The name fields in the table info records for each user data table are initially empty.

The Avigo user :

- Can create a new user table (as long as one of the 8 is unused).
- Can add fields to a user table (as long as not all 32 have been assigned).
- Can change the name of a user data table, and the names of its fields.
- Cannot delete fields from a user table.
- Cannot change the data type of a user table field.
- Cannot create a new user table without at least one field.
- Cannot save a user table with an empty name.

List Records

List records are records for managing lists. Each record corresponds to a list. The number of items in a list, *N*, can range from 0 to 16.

As far as TOPS is concerned, each of the 16 list records always exist. Whether the list name is empty or not determines if the list is visible to the user. To delete a list (from the user's perspective), PUT the list record with an empty name field.

After reset:

- List record 0 has Expense application's **Type** list.
- List record 1 has Expense applications's **Method** list.
- List records 2..15 are deleted (list names are empty).

When sending a record:

- Before deleting a list, change all custom fields and user table fields that refer to that list to the Text128 data type.
- There must be at least one list item in the record if list name is non-empty. (If list name is empty, list items are not required.)
- There must not be any gaps in the sequence of list items. If there are *N* list items, they must occupy the first *N* positions. There must not be any empty list items.
- The list items must be sorted in alphabetical order. (The order given for the list items in the record is the order in which they will be displayed.)
- Lists 0 and 1 are required for the Expense application. Attempts to delete them or rename them are ignored.

The Avigo user:

- Can add lists. Over TOPS, it appears as though a list that previously had an empty name now does have a name and at least one list item.
- Can edit list names (except for the Expense application lists, lists 0 and 1), but cannot leave them empty—so the user cannot delete a list.
- Can delete a list item by editing it and leaving it empty. However, the user is not allowed to save changes to a list unless there is at least one non-empty list item.
- Causes a list record's modified bit to be set when saving changes to the list's name or list items.

Category Label Records

These are records for managing category labels. The number of labels, *N*, can range from 1 to 8. Record X contains the category labels for table X. There are 5 category label records, one for each of the Address, Memo, Schedule, Task, and Expense applications (Table ID's 0..4). The singular case and plural case category titles are presently unused, and should always be given length 0. If received with non-zero length, ignore the title.

The Avigo associates records with categories based on the category label's ID, not its position within the record. The order of the labels within the record determines the order in which they are displayed. This way the PC can modify a category label, resulting in change of position within alphabetical order, without affecting the relationship between categories and their records. Also, deleting a category will not affect records belonging to other categories.

For example, if you PUT a category label record with category labels (Bass, 3), (Drums, 4), (Guitar, 1), and (Unfiled, 0), data records having category ID of 3 will be displayed on the Avigo as belonging to the Bass category.

After reset:

- Each category label record has one category : unfiled.

When sending a record:

- There must be at least one category label, to be used as a default.
- Before sending a category label record that results in category label deletion, you must either delete all records that refer to that ID, or rewrite those records to have a new ID (an ID that exists both before and after the deletion).
- Do not leave any gaps in the sequence of categories in the record: if there are *N* categories they must occupy the first *N* positions.
- The category labels must be sorted in alphabetical order. The order of labels in the record is the order in which they will be displayed.
- Embedded in the most significant bit of the byte for category label ID number is a flag called the owner bit. When this bit is not set, the Avigo user is not considered the owner of the data in that category. The Avigo will not allow the user to edit that data. This is provided for implementing group scheduling.

The Avigo user :

- Can add and edit categories, but not delete them.
- Cannot enter a category label that is the same as any other category label in that application. The Avigo ignores case when comparing category labels.
- Causes the category label record's modified status bit to be set when saving changes to an application's category labels.

T9 Keyboard User Dictionary

This record is used to back up and restore the T9 User Dictionary. A GET returns only the beginning active portion of the 16K flash bank that the T9 User Dictionary resides in. A PUT should only send the data that was received in a prior GET and nothing more. For example, if a GET returns a 4K length dictionary, send only that 4K down to restore, don't pad it up to 16K. The T9 dictionary relies on those unwritten flash bytes (0xff) to expand the dictionary. Before putting this object, you must first use command #3, *Erase T9 User Dictionary*, unless the size of the T9 dictionary before the PUT is 0.

The user dictionary for the T9 keyboard needs to be backed up on a regular basis (weekly, for example). Also, if many new words have been added (over 40), then a backup may be done earlier. This preferably would occur automatically during synchronization without the user having to trigger it. The user dictionary should be restored to the Avigo if the user dictionary becomes corrupted or if the user is replacing a lost or broken Avigo.

Language Bundle

This record is used to replace the existing language bundle with another one. This is usually done to change the Avigo's language. It could also be used to replace the T9 Main Dictionary to one better suited for a particular technical discipline, for example.

After the PUT is finished, the PC should disconnect and then the Avigo resets itself. After the reset, the PC can reconnect to the Avigo over wire.

The user should back up the Avigo data before downloading the language bundle and restore it afterwards. After the language bundle is downloaded, the data in the Avigo is probably unusable, and the user should tap **Yes** when asked whether to configure the unit.

Field Data Types

Each record field in each table is one of the following types. Data types classified as *fixed types* cannot be changed. Data types that are not fixed types can be changed by the PC.

ID	Name	Description
0	Unassigned Type	A type that has not yet been initialized. A record field may originally be defined as this *placeholder* type, giving the user the ability to define and bind the specific type at a later time. As an example, see the custom fields in an Address Table.
1	Text16	A variable-length byte array with a 16-byte maximum length; may be contained in the C type: unsigned char Text16[16];
2	Text128	A variable-length byte array with a 128-byte maximum length; may be contained in the C type: unsigned char Text128[128];
3	Text256	A variable-length byte array with a 256-byte maximum length; may be contained in the C type: unsigned char Text256[256];
4	Text512	A variable-length byte array with a 512-byte maximum length; may be contained in the C type: unsigned char Text512[512];
5	Text3.5K	A variable-length byte array with a 3.5K maximum length; may be contained in the C type: unsigned char Text3_5k[3584];
6	Boolean	1 byte. 0 = False, 1 = True.
7	Integer	2 bytes. Twos complement signed 16-bit integer.
8	Real	8 bytes. IEEE double precision floating point. The first bit is the sign bit, followed by an 11-bit exponent, then a 52-bit mantissa. This type acts as a NumericX type in the Avigo - the number of digits behind the numeric point does not have any restriction.
9	Currency	8 bytes. IEEE double precision floating point. Acts as a Numeric2 type in the Avigo. Only two digits are displayed behind the numeric point.

10	Date	3 bytes.
		typedef struct
		{
		unsigned char YEAR; // YEAR = YEAR - 1901
		unsigned char MONTH; // YEAR range is 1901 ~ 2099
		unsigned char DAY;
		}Date;
		Value 0xffffff represents **No Date**. In schedule record date field, 0x000000 is required in that field for repeating schedules.

11	Time	3 bytes. typedef struct { unsigned char HOUR; unsigned char MINUTE; unsigned char SECOND; }Time; The Avigo will always use 24-hour format to save Time information, and will convert the value for display, if necessary. Value 0xffffff represents **No Time**.

12	ScheduleTime	6 bytes.
		typedef struct {
		Time START_TIME; // See Time definition above.
		Time END_TIME;
		} ScheduleTime;
		Values of all 0xffs represent **No Time**.

13	Alarm	1 byte. Default: 0xff **No Alarm** 0 0 minute 3 15 minutes 6 1 hour 1 5 minutes 4 30 minutes 7 1 hour 30 minutes 2 10 minutes 5 45 minutes 8 2 hours
14	Drawing	3600 bytes. This type is used by the Sketch application. A 3600-byte vector representing a pixel map. Although the Avigo touch panel width and height is 160 * 240 / 8 = 4800 bytes, the Sketch application limits the drawing size to 160 * 180 / 8 = 3600 bytes. The first byte contains the leftmost 8 pixels of the top row, with the corner pixel being in the most significant bit. The following bytes continue to the right on that row and then on to the next row until the end is reached.
15	Category	1 byte. The value range is 0~7.
16	Priority	1 byte. 0xff represents **No Priority**.
17	Repeat (See additional information on page 40A.)	9 bytes. typedef struct { Date StartDate; Date EndDate; unsigned char Flag; unsigned char Data1; unsigned char Data2; }Repeat; Defined in detail below. If Repeat has not been defined, this field is filled with 0xff's. A repeat field may have data in it even if a record is not set to repeat. This happens if the user toys with the repeat field but finally decides not to make it repeat. Whether or not a record is repeating is indicated in another field. For example, in a schedule record, the date field determines this.

ID	Name	Description
18	ListType1	Similar to the data type Text16, except that when the user edits this field the items from List 1 will be shown and the user chooses from them.
19	ListType2	Similar to ListType1, except user chooses from list 2.
20	ListType3	Similar to ListType1, except user chooses from list 3.
21	ListType4	Similar to ListType1, except user chooses from list 4.
22	ListType5	Similar to ListType1, except user chooses from list 5.
23	ListType6	Similar to ListType1, except user chooses from list 6.
24	ListType7	Similar to ListType1, except user chooses from list 7.
25	ListType8	Similar to ListType1, except user chooses from list 8.
26	ListType9	Similar to ListType1, except user chooses from list 9.
27	ListType10	Similar to ListType1, except user chooses from list 10.
28	ListType11	Similar to ListType1, except user chooses from list 11.
29	ListType12	Similar to ListType1, except user chooses from list 12.
30	ListType13	Similar to ListType1, except user chooses from list 13.
31	ListType14	Similar to ListType1, except user chooses from list 14.
32	ListType15	Similar to ListType1, except user chooses from list 15.
33	ListType16	Similar to ListType1, except user chooses from list 16.
34	Private	This data type is only used internally by the handheld. Private status of a record is communicated in the status bits of a data table record.
35	TaskDate	Used in tasks. If a task is **single**, this is the same as the data type Date and is 3 bytes long. If the task is **repeat**, this becomes a list of up to 100 Dates with 3 bytes for each date.
36	CompleteDate	Same as TaskDate, but has a checkbox for user to indicate task is completed.
37	Synchronizer Key	A variable-length byte array with a-128 byte maximum length. A general-purpose field, hidden from the user, useful for facilitating synchronization with PC PIMs.
38	Custom Field Integers	8 bytes. This has the same internal format as real type (data type #8). When displayed, it does not show the decimal point or digits to the right of that. This is used for the integer type for custom fields and data function fields. PC must truncate any fractional part of the number before sending it to handheld. Handheld does not permit user to enter a fractional part into this field.

Unused data type IDs 39..127 are reserved for future use.

Downloadable applications may freely use data type IDs 128..255. The maximum number of fields a downloadable application can define for its records is 32, and the total max size of the combined fields may not be greater than 4400 bytes.

Additional Notes on the Repeat Field Type

Flag field

D7	D6	D5	D4	D3	D2	D1	D0

Low four bits: = 1 (0001b) : set by day
= 2 (0010b) : set by week
= 3 (0011b) : set by month
= 4 (0100b) : set by year

D3~D6: Reserved

D7 = 0 : select day (by month)
= 1 : select date (by month)

Data1 and Data2 fields

- If Flag = 00000001b (set by day), repeats every x days.
  Data2 value range is 1~28.
- If Flag = 00000010b (set by week), repeats every week
  Data2 definition is as follows:

D7	D6	D5	D4	D3	D2	D1	D0

D0 = 1 → every Sunday
D1 = 1 → every Monday
D2 = 1 → every Tuesday
D3 = 1 → every Wednesday
D4 = 1 → every Thursday
D5 = 1 → every Friday
D6 = 1 → every Saturday

- If Flag = 00000011b (set by month and select day mode), repeats on day x of week y of each month.

Data1: = 1 → first week
= 2 → second week
= 3 → third week
= 4 → fourth week
= 5 → last week

Data2: = 0 → Sunday
= 1 → Monday
= 2 → Tuesday
= 3 → Wednesday
= 4 → Thursday
= 5 → Friday
= 6 → Saturday

- If Flag = 10000011b (set by month and select date mode),
  repeat on day x of every month.
  Data2 value range is 0~31.
  If Data2 equals 0, it means the last day of the month.

- If Flag = 00000100b (set by year),
  repeat yearly on day x of month y
  Data1 indicates month.
  Data2 indicates day.

Sys Info Items: Category A Read/Write

In addition to being read and written individually, all the Category A items may be read by the PC with a single GET of the Category A Grouped Object (Object Type 2, Subtype 7).

Sys Info Name	**Sys Info #**	**Data**
Database Status	0	16 bytes. Avigo should not modify or internally make use of this data. When handheld is reset (flash initialized), each byte should be reset to 0x00. Database Status is used by the PC program to check whether the handheld has been synchronized with some other PC.
PC Storage Area for Category A	1	Additional 64 bytes for PC's use. Avigo should not modify or internally make use of this data. When handheld is reset (flash initialized), each byte should be reset to 0x00.
Power-on Message.	7	Text to be displayed on the power-on screen. If a string is not long enough to fill the entire field, the byte following the end of the string is 0x00. Keep in mind that if the string does fill the entire field, then there will be no 0x00 byte. 32 bytes : Name 160 bytes : Note
Automatic Optimization Settings	8	Byte 1 : 0 = manually 1 = daily 2 = weekly Byte 2 : Hour (0..23) Byte 3 : Day of week (0..6 where 0 is Sunday)

Calendar Application Preferences	10	Byte 1: Start the day at: (0..18) Byte 2: Event Time Style: 0 = Start time and end time 1 = Start time and duration Byte 3: Default Duration: 0 = 15 minutes 1 = 30 minutes 2 = 1 hour Byte 4: Time Slot 0 = 15 minutes 1 = 30 minutes 2 = 1 hour Byte 5: Advance Notice: 0 = 0 minutes 1 = 5 minutes 2 = 10 minutes 3 = 15 minutes 4 = 30 minutes 5 = 45 minutes 6 = 1 hour 7 = 1 hour, 30 minutes 8 = 2 hours
Task Application Preferences	11	Byte 1: Maximum Alpha Priority: 1 = No numeric priority 2 = B 3 = C 4 = D 5 = E Byte 2: Maximum Numeric Priority 1 = No numeric priority 2 = 2 3 = 3 … 9 = 9 Byte 3: Reserved for future use. Always 0. When PUTting this SysInfo item, set it to 0. Byte 4: Reserved for future use. Always 0. When PUTting this SysInfo item, set it to 0.

Address Application Preferences	12	Byte 1: Address Format: 0 = North American 1 = European

Sys Info Items: Category A Read Only

In addition to being read and written individually, all the Category A items may be read by the PC with a single GET of the Category A Grouped Object (Object Type 2, Subtype 7).

Sys Info Name	**Sys Info #**	**Data**
Memory Available	2	4 bytes : Available Total # Bytes available for storage without optimizing. 4 bytes : # Bytes taken up by deleted objects. 4 bytes : # Bytes taken up by active objects
Language Bundle	6	1 byte: 0: No language bundle loaded (not possible currently) 1: English 2: German 3: French 4: Spanish 5: Italian

Sys Info Items: Category B Read/Write

Format is the same as for Category A Read/Write items, but these items may not be read as a group.

Handheld Identifier	0	16 bytes. Avigo should not modify or internally make use of this data because it may have different meanings for different PIMS used with the Avigo. When handheld is reset (flash initialized), each byte should be reset to 0x00. Handheld ID is used by the PC program to determine which handheld this is.
PC Storage Area for Category B	1	Additional 64 bytes for PC's use. Avigo should not modify or internally make use of this data. When unit is reset (flash initialized), each byte should be reset to 0x00.
Synchronization Screen Message	3	64 bytes. Text formatted to be shown on synchronization screen after Avigo receives a 'disconnect' command or the connection is lost. This acts like a synchronization log, but is much shorter, so you might call it a synchronization summary. This message can be an error message sent from the PC so that Avigo can show it next time the user goes to the synchronization screen (in case the synchronization process is unexpectedly terminated). It can also be a message saying the synchronization process is successfully completed. If the text does not fill the entire field, then the byte following the text is 0x00 (any remaining bytes after the 0x00 may have any value). Keep in mind that if the string does fill the entire field, then there will be no 0x00 byte. If text is longer than one line, carriage returns can be used to format text on display. This message is also set by the PC through the use of the Display Message command (#2). Default is empty.
Splash Screen Image	4	4800 bytes: The image is a sequence of 240 rows. The first row is the top line of the image. Each row is 20 bytes where the most significant bit of the first byte is the leftmost pixel of the row.

Sys Info Items: Category C Read/Write

Category C Read/Write consists of system information and settings that can be read and written through TOPS only if the handheld is unlocked. They are of the same format as Category A Read/Write Objects.

Sys Info Name	Sys Info #	Data
Password	0	8 bytes. For passwords shorter than 8 bytes, the byte following the password is 0x00. (Any remaining bytes after the 0x00 may have any value.) Keep in mind that if the string does fill the entire field, then there will be no 0x00 byte. An unset password has 0x00 as the first byte. This item cannot be written without first using the unlock command if the password has been set.

Sys Info Items: Category D

These items can be read and written through TOPS, but they may not be gotten as a group. Sys info items in this category may require parameters or may be used to alert the handheld of some upcoming action.

Sys Info Name	Sys Info #	Capability	Parameters	Data
New Record ID	0	Read only	2 bytes: size of record 2 bytes : ID of table for new record	4 bytes
New Application ID	1	Read only	2 bytes: App size in Kb (round up to the nearest kilobyte).	2 bytes

New IDs for New Record ID and New Application ID are used by the PC to find out what the next ID should be when it sends a new object to the handheld. When the new record is created in the handheld with the new ID, another ID will then become the current new ID.

Creating New Data Records

To create new data records, you should GET the New Record ID Sys Info Item (Category D, item 0) to get an ID for the record, and then PUT the record using that ID. The handheld returns IDs that minimize the size of its index tables. It is also possible to PUT a new record having a previously unused ID of your choice—just be sure it really is an unused ID. This way, when restoring a backup file you can PUT the records with the same IDs that they had before if that is important for synchronization. Using the New Record ID Sys Info item is preferred.

To calculate the size of a record for the purpose of requesting a new record ID:

9 + (num_fields * 2) + field_size (field 0) + field_size (field 1) + ... + field_size (field n-1)

The num_fields parameter is the maximum number of available fields for the record. For example, for addresses it is 34 and for user data tables it is 33.

When you define a new custom field on the Avigo, all records created or edited before will not have that field, but all records created or edited after will have that field whether it is empty or not.

If custom fields are defined for a table, you do not have to include them when you PUT the data record to the Avigo, even if they are fixed-length fields. Fixed-type fields, those non-custom fields that cannot have their data type changed, must be included in the PUT of a record if their size is fixed-length.

If the PC is going to change the data type of a custom field or delete a custom field, it is preferable to follow this procedure:

1. Delete all records in the table.
2. PUT the new field definition record.
3. Rewrite all the records with fields corrected for the new field definition.

(This is called a “wash.” See page 15 for more information.)

It may be possible to rewrite just a subset of the data records in a table. When the Avigo stores a data record, it includes all fields, whether empty or not, up to the last assigned custom field. So when deleting a custom field, the PC must be certain to rewrite any records for which that field is empty but present.

If that present-but-empty field is left there and the field is later assigned as a fixed-length data type, the Avigo may not handle it well. Also, during a field type change, it is necessary to rewrite these present-but-empty fields if it has been redefined to a fixed-length type. You also have to be careful of these empty-but-present fields when deleting a custom field that is not last and you are shifting down the fields that follow after the deleted one. Not only do you have to rewrite the ones having data in fields after the deleted field, but you have to rewrite those with present-but-empty fields if the new data type for that field is fixed-length instead of variable length. So, in general, when trying to economize on rewrites, watch out!

Appendix B: Header Files

tops_pda.h

```
#ifndef TOPS_PDA_H
#define TOPS_PDA_H

#ifdef __cplusplus
extern "C" {
#endif

/*****************************************************************************\
*  tops_pda.h
*
*  Header file that defines the data types and functions for the TOPS interface
*  to the PDA-100.
*  TOPS Tables consist of records, and records consist of fields.
*  This header file defines the format of all the fields.
*  Then it defines the records that are made up of those fields, and finally
*  it defines the functions that are used to send and receive the records.
*
\*****************************************************************************/

// define options
#ifdef __BORLANDC__
#define DLLEXPORT __export _stdcall
// Borland pragma for assuring byte alignment
#pragma option -a1
#else
#define DLLEXPORT __declspec(dllexport) _stdcall
#define _USERENTRY _cdecl
// MFC pragma's for assuring byte alignment
#pragma pack( push, alignment_before_tops_includes ) //store current alignment
#pragma pack(1)                                      //set up byte alignment
#endif


typedef unsigned long  DWORD;
typedef int            BOOL;
typedef unsigned char  BYTE;
typedef unsigned short WORD;


/*****************************************************************************
// Field Types.
*****************************************************************************/

typedef struct TOPS_UnassignedDef
{
  WORD FieldLength;
  BYTE Fields[128];
```

```
} TOPS_Unassigned;


typedef struct TOPS_Text8Def
{
  WORD FieldLength;
  BYTE Text[8];
} TOPS_Text8;


typedef struct TOPS_Text16Def
{
  WORD FieldLength;
  BYTE Text[16];
} TOPS_Text16;

typedef struct TOPS_Text127Def
{
  BYTE FieldLength;
  BYTE Text[127];
} TOPS_Text127;


typedef struct TOPS_Text128Def
{
  WORD FieldLength;
  BYTE Text[128];
} TOPS_Text128;

typedef struct TOPS_Text255Def
{
  BYTE FieldLength;
  BYTE Text[255];
} TOPS_Text255;

typedef struct TOPS_Text256Def
{
  WORD FieldLength;
  BYTE Text[256];
} TOPS_Text256;

typedef struct TOPS_Text512Def
{
  WORD FieldLength;
  BYTE Text[512];
}  TOPS_Text512;

typedef struct TOPS_Text3_5KDef
{
  WORD FieldLength;
  BYTE Text[3584];
}  TOPS_Text3_5K;
```

```
typedef BYTE   TOPS_Boolean;
typedef WORD   TOPS_Integer;
typedef BYTE   TOPS_Byte;
typedef WORD   TOPS_Word;
typedef double TOPS_Real;
typedef double TOPS_Currency;
typedef double TOPS_IntegerCustomField;
typedef struct TOPS_DateDef  // definition of data and time
{
  BYTE Year;
  BYTE Month;
  BYTE Day;
}TOPS_Date;


typedef struct TOPS_TimeDef
{
  BYTE Hour;
  BYTE Minute;
  BYTE Second;
}TOPS_Time;

typedef struct TOPS_TaskDateDef
{
  WORD FieldLength;
  TOPS_Date Date[100];
}TOPS_TaskDate;

typedef struct TOPS_ScheduleTimeDef
{
  TOPS_Time StartTime;
  TOPS_Time EndTime;
}TOPS_ScheduleTime;
typedef BYTE TOPS_Alarm;    // 0-reserved, 0=0 min, 1=5,2=10,3=15,4=30,5=45,
                            // 6=1 hour, 7=1.5, 8=2, 0xff=no alarm
typedef struct
{  WORD FieldLength;
   BYTE Drawing[3600];
} TOPS_Drawing;
typedef BYTE TOPS_Category; //range 0-7
typedef BYTE TOPS_Priority; // range 1-3, 0xff=no priority
typedef struct TOPS_RepeatDef
{
  TOPS_Date StartDate;
  TOPS_Date EndDate;
  BYTE Flag;
  BYTE Data1;
  BYTE Data2;
}TOPS_Repeat;
typedef struct TOPS_ListTypeDef
{
  BYTE  FieldLength;
  BYTE Text[16];
```

```
}TOPS_ListFieldType;
typedef TOPS_ListFieldType TOPS_ListType1;
typedef TOPS_ListFieldType TOPS_ListType2;
typedef TOPS_ListFieldType TOPS_ListType3;
typedef TOPS_ListFieldType TOPS_ListType4;
typedef TOPS_ListFieldType TOPS_ListType5;
typedef TOPS_ListFieldType TOPS_ListType6;
typedef TOPS_ListFieldType TOPS_ListType7;
typedef TOPS_ListFieldType TOPS_ListType8;
typedef TOPS_ListFieldType TOPS_ListType9;
typedef TOPS_ListFieldType TOPS_ListType10;
typedef TOPS_ListFieldType TOPS_ListType11;
typedef TOPS_ListFieldType TOPS_ListType12;
typedef TOPS_ListFieldType TOPS_ListType13;
typedef TOPS_ListFieldType TOPS_ListType14;
typedef TOPS_ListFieldType TOPS_ListType15;
typedef TOPS_ListFieldType TOPS_ListType16;
typedef TOPS_Date TOPS_DueDate;
typedef TOPS_Date TOPS_RepeatDueDate[100];
typedef TOPS_Byte TOPS_FieldDefName[16];
typedef TOPS_Date TOPS_AdvanceNotice;
typedef struct
{
  WORD FieldLength;
  BYTE Key[128];
}TOPS_SynchronizerKey;
typedef struct TOPS_IconDef
{
  WORD IconLength;
  BYTE Icon[5*1024];
} TOPS_Icon;
typedef struct TOPS_CatLabelTypeDef
{
  BYTE  FieldLength;
  BYTE Text[16];
}TOPS_CategoryLabelType;

#define MAX_LIST_NAME_LENGTH 16
typedef struct TOPS_ListNameDef
{
  BYTE  FieldLength;  // 0..16  (The name can be at most 16 bytes)
  BYTE  Text[MAX_LIST_NAME_LENGTH];
}TOPS_ListName;

#define MAX_NUM_OF_ITEMS_IN_LIST 16
#define MAX_SIZE_OF_ITEM_NAME    16
typedef struct TOPS_ListItemDef
{
  BYTE FieldLength;
  BYTE Text[MAX_SIZE_OF_ITEM_NAME];
}TOPS_ListItems;

typedef struct TOPS_ApplicationRecordDef
```

```
{
  DWORD Length;   // Apps size can be DWORD (which could be huge),
  BYTE Binary[];  // so don't use max size.
} TOPS_Application;


typedef struct TOPS_TableStatusRecordDef
{
  WORD  Length;
  BYTE  Data[128];
} TOPS_TableStatus;

typedef struct TOPS_DictionaryRecordDef
{
  DWORD Length;
  BYTE  Data[];
} TOPS_Dictionary;

typedef struct TOPS_LanguageBundleRecordDef
{
  DWORD Length;
  BYTE  Data[];
} TOPS_LanguageBundle;

typedef struct TOPS_PatchRecordDef
{
  DWORD Length;
  BYTE  Data[65535];
} TOPS_Patch;

typedef struct TOPS_ApplicationPreferenceRecordDef
{
  WORD Length;
  BYTE Data[256];
} TOPS_ApplicationPreference;

// the following structure defines the format in which data records are
// actually sent to the handheld.  This is the format that would be seen
// on the wire connecting the PC to the handheld.
//
typedef struct TOPS_GenericRecordFieldDef
{
  WORD FieldId;    // Field Id
  WORD DataLength; // Length of Field Data
  BYTE Data[];     // Data (size=DataLength)
} TOPS_GenericRecordField;
```

```
/*****************************************************************************
// Define the records of the differenct Record Tables. There are several
// different Record Tables in the PDA-100.  Each Record Table has a different
// format for its record.  The Record Tables are:
//
// 1. Address Table
// 2. Memo Table
// 3. Schedule Table
// 4. Task Table
// 5. Expense Table
// 6. Sketch Table
// 7. Data Tables(8)
// 8. Custom Downloadable App Record
//
*****************************************************************************/

typedef struct TOPS_AddressRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Category        Category;
  TOPS_Text128         LastName;
  TOPS_Text128         FirstName;
  TOPS_Text128         Company;
  TOPS_Text128         Title;
  TOPS_Text128         BusinessPhone;
  TOPS_Text128         Fax;
  TOPS_Text128         HomePhone;
  TOPS_Text128         Cellular;
  TOPS_Text128         Pager;
  TOPS_Text128         Email;
  TOPS_Text128         BusinessStreet;
  TOPS_Text128         BusinessCity;
  TOPS_Text128         BusinessState;
  TOPS_Text128         BusinessZip;
  TOPS_Text128         BusinessCountry;
  TOPS_Text128         HomeStreet;
  TOPS_Text128         HomeCity;
  TOPS_Text128         HomeState;
  TOPS_Text128         HomeZip;
  TOPS_Text128         HomeCountry;
  TOPS_Date            Birthday;
  TOPS_Text512         Notes;
  TOPS_Integer         IndexBy;
  TOPS_Integer         PrimaryComm;
  TOPS_Unassigned      Custom1;
  TOPS_Unassigned      Custom2;
  TOPS_Unassigned      Custom3;
  TOPS_Unassigned      Custom4;
  TOPS_Unassigned      Custom5;
  TOPS_Unassigned      Custom6;
  TOPS_Unassigned      Custom7;
  TOPS_Unassigned      Custom8;
}TOPS_AddressRecord;
```

```
typedef struct TOPS_MemoRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Category        Category;
  TOPS_Text256         Title;
  TOPS_Text3_5K        Memo;
  TOPS_Unassigned      Custom1;
  TOPS_Unassigned      Custom2;
  TOPS_Unassigned      Custom3;
  TOPS_Unassigned      Custom4;
}TOPS_MemoRecord;

typedef struct TOPS_ScheduleRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Category        Category;
  TOPS_Date            Date;
  TOPS_ScheduleTime    Time;
  TOPS_Text256         Description;
  TOPS_Repeat          Repeat;
  TOPS_Alarm           Alarm;
  TOPS_Text512         Note;
  TOPS_Unassigned     Custom1;
  TOPS_Unassigned     Custom2;
  TOPS_Unassigned     Custom3;
  TOPS_Unassigned     Custom4;
  TOPS_Unassigned     Custom5;
  TOPS_Unassigned     Custom6;
  TOPS_Unassigned     Custom7;
  TOPS_Unassigned     Custom8;
}TOPS_ScheduleRecord;

typedef struct TOPS_TaskRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Category        Category;
  TOPS_TaskDate        StartDate;
  TOPS_TaskDate        DueDate;
  TOPS_TaskDate        CompleteDate;
  TOPS_Text256         Description;
  TOPS_Priority        Priority;
  TOPS_Repeat          Repeat;
  TOPS_Text512         Note;
  TOPS_Unassigned     Custom1;
  TOPS_Unassigned     Custom2;
  TOPS_Unassigned     Custom3;
  TOPS_Unassigned     Custom4;
  TOPS_Unassigned     Custom5;
  TOPS_Unassigned     Custom6;
  TOPS_Unassigned     Custom7;
  TOPS_Unassigned     Custom8;
```

```
}TOPS_TaskRecord;


typedef struct TOPS_ExpenseRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Category        Category;
  TOPS_Date            DueDate;
  TOPS_Currency        Amount;
  TOPS_ListType1       Type;
  TOPS_Text128         Description;
  TOPS_Text128         Merchant;
  TOPS_Text128         Location;
  TOPS_ListType2       Method;
  TOPS_Text512         Note;
  TOPS_Unassigned      Custom1;
  TOPS_Unassigned      Custom2;
  TOPS_Unassigned      Custom3;
  TOPS_Unassigned      Custom4;
  TOPS_Unassigned      Custom5;
  TOPS_Unassigned      Custom6;
  TOPS_Unassigned      Custom7;
  TOPS_Unassigned      Custom8;
}TOPS_ExpenseRecord;

typedef struct TOPS_SketchRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Integer         ID;
  TOPS_Text16          Title;
  TOPS_Drawing         Drawing;
}TOPS_SketchRecord;

typedef struct TOPS_DataTableRecordDef
{
  TOPS_SynchronizerKey SynchronizerKey;
  TOPS_Unassigned     Custom1;
  TOPS_Unassigned     Custom2;
  TOPS_Unassigned     Custom3;
  TOPS_Unassigned     Custom4;
  TOPS_Unassigned     Custom5;
  TOPS_Unassigned     Custom6;
  TOPS_Unassigned     Custom7;
  TOPS_Unassigned     Custom8;
  TOPS_Unassigned     Custom9;
  TOPS_Unassigned     Custom10;
  TOPS_Unassigned     Custom11;
  TOPS_Unassigned     Custom12;
  TOPS_Unassigned     Custom13;
  TOPS_Unassigned     Custom14;
  TOPS_Unassigned     Custom15;
  TOPS_Unassigned     Custom16;
  TOPS_Unassigned     Custom17;
```

```
  TOPS_Unassigned     Custom18;
  TOPS_Unassigned     Custom19;
  TOPS_Unassigned     Custom20;
  TOPS_Unassigned     Custom21;
  TOPS_Unassigned     Custom22;
  TOPS_Unassigned     Custom23;
  TOPS_Unassigned     Custom24;
  TOPS_Unassigned     Custom25;
  TOPS_Unassigned     Custom26;
  TOPS_Unassigned     Custom27;
  TOPS_Unassigned     Custom28;
  TOPS_Unassigned     Custom29;
  TOPS_Unassigned     Custom30;
  TOPS_Unassigned     Custom31;
  TOPS_Unassigned     Custom32;
} TOPS_DataTableRecord;


/****************************************************************************
// Define the Custom Downloadable App record.  Downloadable Apps can have a
// data table associated with them.  This data table must have a table id
// in the range 128 to 256.  This comment record defines the format for those
// data tables.
****************************************************************************/

//typedef struct TOPS_CustomRecordDef
//{
//  WORD Length;
//  TOPS_GenericRecordField Field[];  // an array of fields
//}TOPS_CustomRecord;



typedef union TOPS_RecordTableRecordDef
{
  TOPS_AddressRecord          AddressRecord;
  TOPS_MemoRecord             MemoRecord;
  TOPS_ScheduleRecord         ScheduleRecord;
  TOPS_TaskRecord             TaskRecord;
  TOPS_ExpenseRecord          ExpenseRecord;
  TOPS_SketchRecord           SketchRecord;
  TOPS_DataTableRecord        DataTableRecord;
} TOPS_RecordTableRecords;


/****************************************************************************
// Table Info Table record definition.
// There is only 1 Table Info Table.
// It is used to get/put information about any table in the PDA-100.
// For the Table Info Table the record id corresponds to a table id. For
// example, reading record 1 will get the table info for the Memo Table since
// the Memo Tables Table Id is 1.
****************************************************************************/
```

```
typedef struct TOPS_RecordObjectsTableInfoRecordDef
{
  TOPS_Word Reserved;    // Reserved value, always = 0
  TOPS_Byte ReadOnly;    // 0=Modifiable, 1=ReadOnly
  TOPS_Text255 AppName;  // Name of Application
} TOPS_TableInfoRecord;

/*****************************************************************************
// Field Definition Table record definition.
// There is only 1 Field Definition Table.
// It is used to get/put the field definitions for the records of a table
// in the PDA-100.  The table id is constant (0x8006).
//  The record id corresponds to the table this field definition defines.
*****************************************************************************/

typedef enum
{
  TOPS_UnassignedFieldType   = 0,
  TOPS_Text16FieldType       = 1,
  TOPS_Text128FieldType      = 2,
  TOPS_Text256FieldType      = 3,
  TOPS_Text512FieldType      = 4,
  TOPS_Text3_5KFieldType     = 5,
  TOPS_BooleanFieldType      = 6,
  TOPS_IntegerFieldType      = 7,
  TOPS_RealFieldType         = 8,
  TOPS_CurrencyFieldType     = 9,
  TOPS_DateFieldType         = 10,
  TOPS_TimeFieldType         = 11,
  TOPS_ScheduleTimeFieldType = 12,
  TOPS_AlarmFieldType        = 13,
  TOPS_DrawingFieldType      = 14,
  TOPS_CategoryFieldType     = 15,
  TOPS_PriorityFieldType     = 16,
  TOPS_RepeatFieldType       = 17,
  TOPS_ListType1FieldType    = 18,
  TOPS_ListType2FieldType    = 19,
  TOPS_ListType3FieldType    = 20,
  TOPS_ListType4FieldType    = 21,
  TOPS_ListType5FieldType    = 22,
  TOPS_ListType6FieldType    = 23,
  TOPS_ListType7FieldType    = 24,
  TOPS_ListType8FieldType    = 25,
  TOPS_ListType9FieldType    = 26,
  TOPS_ListType10FieldType   = 27,
  TOPS_ListType11FieldType   = 28,
  TOPS_ListType12FieldType   = 29,
  TOPS_ListType13FieldType   = 30,
  TOPS_ListType14FieldType   = 31,
  TOPS_ListType15FieldType   = 32,
  TOPS_ListType16FieldType   = 33,
  TOPS_PrivateFieldType      = 34,
```

```
  TOPS_TaskDateFieldType     = 35,
  TOPS_CompleteDateFieldType = 36,
  TOPS_SynchronizerKeyFieldType = 37
}TOPS_DataFieldType;

typedef struct TOPS_RecordObjectsFieldDefinitionDef
{
  TOPS_Byte         DataType;  // TOPS_DataFieldTypes (used TOPS_Byte because C
                               // enumerated type is a WORD not BYTE).
  TOPS_Byte         FixedFlag; // 0 = fixed, 1 = type can be changed by PC
  TOPS_FieldDefName Name;      // name of field
} TOPS_FieldDefinition;

typedef struct TOPS_RecordObjectsFieldDefinitionRecordDef
{
  DWORD Length;
  TOPS_FieldDefinition Fields[];  // array of fields
}TOPS_FieldDefinitionRecord;

/*****************************************************************************
// Application Record Table record definition.
// There is only 1 Application Record Table.
// It is used to get/put the data record of an application in the PDA-100.
// For the Application Record Table the table id is constant (0x8001).
// The record id corresponds to the Application Id.
// (All app records must have App Info defined at the beginning of the data.
//  App Info is read by doing a get of the App Info Table, App Info is set
//  by doing a put of the App Record Table.)
*****************************************************************************/

typedef struct TOPS_AppRecDef
{
  TOPS_Application Application;
}TOPS_ApplicationRecord;

/*****************************************************************************
// Application Info Table record definition.
// There is only 1 Application Info Record Table.
// It is used to get the info of an application in the PDA-100.
// For the Field Definition Table the table id is constant (0x8002).
// The record id corresponds to the Application Id.
*****************************************************************************/

#define MAX_APPLICATION_NAME_LENGTH      256
#define MAX_APPLICATION_VERSION_LENGTH   128
#define MAX_APPLICATION_ICON_LENGTH      5*1024
#define MAX_APPLICATION_COPYRIGHT_LENGTH 128

typedef struct TOPS_ApplicationInfoRecordDef
{
  TOPS_Byte       AppType;
  TOPS_Byte       DataTableFlag;
  TOPS_Date       CreationDate;
```

```
  TOPS_Time       CreationTime;
  TOPS_Word       Checksum;
  TOPS_Text255    Name;
  TOPS_Text255    Version;
  TOPS_Icon       Icon;
  TOPS_Text127    Copyright;
  TOPS_Unassigned Reserved2;
} TOPS_ApplicationInfoRecord;

/*****************************************************************************
// Application Preference Table record definition.
// There is only 1 Application Record Table.
// It is used to get/put the preference record of an application in the PDA-100.
// For the Application Preference Table the table id is constant (0x8005).
// The record id corresponds to the Table Id
*****************************************************************************/

typedef struct TOPS_AppPrefRecDef
{
  TOPS_ApplicationPreference Preference;
}TOPS_ApplicationPreferenceRecord;

/*****************************************************************************
// PDA List Table record definition.
// There is only 1 PDA List Table.
// It is used to get/set list info in the PDA-100.
// For the PDA List Table the table id is constant (0x8003).
// The record id corresponds to the List Record Id which ranges from 0 to 15.
*****************************************************************************/

typedef struct TOPS_ListRecordDef
{
  TOPS_ListName  ListName;
  TOPS_ListItems Item1;
  TOPS_ListItems Item2;
  TOPS_ListItems Item3;
  TOPS_ListItems Item4;
  TOPS_ListItems Item5;
  TOPS_ListItems Item6;
  TOPS_ListItems Item7;
  TOPS_ListItems Item8;
  TOPS_ListItems Item9;
  TOPS_ListItems Item10;
  TOPS_ListItems Item11;
  TOPS_ListItems Item12;
  TOPS_ListItems Item13;
  TOPS_ListItems Item14;
  TOPS_ListItems Item15;
  TOPS_ListItems Item16;
} TOPS_ListRecord;
```

```
/*****************************************************************************
// Category Label Table record definition.
// There is only 1 Category Label Table.
// It is used to get/set categories in the PDA-100.
// For the Category Label Table the table id is constant (0x8004).
// The record id corresponds to the table id whose category labels are desired.
*****************************************************************************/

typedef struct TOPS_CategoryLabelRecordDef
{
  TOPS_CategoryLabelType SingularTitle;
  TOPS_CategoryLabelType PluralTitle;
  TOPS_CategoryLabelType Label1Name;
  TOPS_Byte              Label1Id;
  TOPS_CategoryLabelType Label2Name;
  TOPS_Byte              Label2Id;
  TOPS_CategoryLabelType Label3Name;
  TOPS_Byte              Label3Id;
  TOPS_CategoryLabelType Label4Name;
  TOPS_Byte              Label4Id;
  TOPS_CategoryLabelType Label5Name;
  TOPS_Byte              Label5Id;
  TOPS_CategoryLabelType Label6Name;
  TOPS_Byte              Label6Id;
  TOPS_CategoryLabelType Label7Name;
  TOPS_Byte              Label7Id;
  TOPS_CategoryLabelType Label8Name;
  TOPS_Byte              Label8Id;
}TOPS_CategoryLabelRecord;

/*****************************************************************************
// Table Status Table record definition.
// There is only 1 Application Record Table.
// It is used to get/put synchronization status of data tables.
// For the Table Status Table the table id is constant (0x8007).
// The record id corresponds to the Table Id.
*****************************************************************************/

typedef struct TOPS_TableStatusRecDef
{
  TOPS_TableStatus Status;
}TOPS_TableStatusRecord;

/*****************************************************************************
// Dictionary Table record definition.
// There is only 1 Dictionary Record Table.
// It is used to get/put the JustType User Dictionary.
// For the Dictionary Table the table id is constant (0x8008).
*****************************************************************************/

typedef struct TOPS_DictionaryRecDef
{
  TOPS_Dictionary Dictionary;
```

```
}TOPS_DictionaryRecord;

/*****************************************************************************
// Language Bundle Table record definition.
// There is only 1 Language Bundle Record Table.
// It is used to get/put the language specific data including the
// JustType Main Dictionary.
// For the Language Bundle Table the table id is constant (0x8009).
*****************************************************************************/

typedef struct TOPS_LanguageBundleRecDef
{
  TOPS_LanguageBundle LanguageBundle;
}TOPS_LanguageBundleRecord;
```

Note:
Patch Table is no longer a valid table on the Avigo.

```
/*****************************************************************************
// Patch Table record definition.
// There is only 1 Patch Record Table.
// It is used to put a patch to the PDA.
// For the Patch Table the table id is constant (0x800A).
*****************************************************************************/

typedef struct TOPS_PatchRecDef
{
  TOPS_Patch Patch;
}TOPS_PatchRecord;



/*****************************************************************************
// The union of all possible TOPS record types.
*****************************************************************************/


typedef union TOPS_RecordDef
{
  TOPS_FieldDefinitionRecord  FieldDefinition;
  TOPS_RecordTableRecords     RecordTable;
  TOPS_TableInfoRecord        TableInfo;
  TOPS_ListRecord             ListTable;
  TOPS_ApplicationInfoRecord  AppInfoTable;
  TOPS_ApplicationRecord      AppTable;
  TOPS_ApplicationPreferenceRecord AppPrefTable;
  TOPS_CategoryLabelRecord    CategoryLabelTable;
  TOPS_TableStatusRecord      TableStatus;
  TOPS_DictionaryRecord       DictionaryTable;
  TOPS_LanguageBundleRecord   LanguageBundleTable;
  TOPS_Patch                  PatchTable;
} TOPS_Record;


//
```

```
// TOPS API Function Prototypes Overview
//
// The following function prototypes define the interface of the
// Table Oriented Protocol for Synchronization (TOPS) API.
// The TOPS API provides the data structure definitions and the functions
// necessary to communicate with the PDA-100.  The TOPS API uses the I/O API
// for handling the lower level protocol details.
//
// As the name implies the TOPS protocol is Table Oriented.  This means that
// all data communication to the PDA-100 is via a Table. Different tables are
// accessed by their associated Table Id.   Tables consist of one or more
// records.  The records in the tables can be accessed via Record Id's.

// TOPS is the protocol layer above the OBEX layer.
//
// All functions in this API return TRUE, FALSE, or UNEXPECTED RESPONSE.
// If the function returns TRUE, the function call was successful.
// If the function returns FALSE, the function call was unsuccessful.
// If the function returns UNEXPECTED RESPONSE then the PDA-100 is reporting
// an error and a call to TOPS_ReadUnexpectedResponse should be made to get
// the error information.
// Additional error information can be retrieved by examining the I/O API's
// Error Buffer.  The Error Buffer is a circular buffer that is written to
// by the software whenever an error occurs.  The include file IO_ERRORS.H
// contains more information about the I/O API Error Buffer.
//

// Function values returned by TOPS API Functions.
typedef enum TOPS_ReturnValuesDef
{
   TOPS_Failure                = 0,
   TOPS_Success                = 1,
   TOPS_UnexpectedResponse     = 2
}TOPS_ReturnValues;

// Table Types used as parameter on API calls.

typedef enum TOPS_TableTypesDef
{
   // TOPS record tables can range from 0-0x3fff.  Tables 0-15 are defined for
   // all handhelds.
   TOPS_FirstRecordTable       = 0,    // lowest record table #
   TOPS_AddressTable           = 0,    // address table is table 0
   TOPS_MemoTable              = 1,    // memo table is is table 1
   TOPS_ScheduleTable          = 2,
   TOPS_TaskTable              = 3,
   TOPS_ExpenseTable           = 4,
   TOPS_SketchTable            = 5,
   TOPS_Reserved1              = 6,
   TOPS_Reserved2              = 7,
   TOPS_Data1Table             = 8,
   TOPS_Data2Table             = 9,
   TOPS_Data3Table             = 10,
```

```
   TOPS_Data4Table             = 11,
   TOPS_Data5Table             = 12,
   TOPS_Data6Table             = 13,
   TOPS_Data7Table             = 14,
   TOPS_Data8Table             = 15,
   TOPS_FirstCustomAppTable    = 128,
   TOPS_LastCustomAppTable     = 256,
   TOPS_LastRecordTable        = 0x3fff, // highest record table #
   // Non-Record Tables
   TOPS_TableInfoTable         = 0x8000,
   TOPS_ApplicationRecordTable = 0x8001,
   TOPS_ApplicationInfoTable   = 0x8002,
   TOPS_ListTable              = 0x8003,
   TOPS_CategoryLabelTable     = 0x8004,
   TOPS_ApplicationPreferenceTable = 0x8005,
   TOPS_FieldDefTable          = 0x8006,
   TOPS_TableStatusTable       = 0x8007,
   TOPS_DictionaryTable        = 0x8008,
   TOPS_LanguageBundleTable    = 0x8009,
   TOPS_PatchTable             = 0x800A,
   TOPS_LastTable
}TOPS_TableType;


// Definition of the bits in the status words of the records of a table.
typedef enum TOPS_StatusDef
{
   TOPS_Private                = (1<<8),
   TOPS_FieldDefModified       = (1<<5),
   TOPS_DateModified           = (1<<4),
   TOPS_DeleteRecordTag        = (1<<3), //(Deletes record too if it exists)
   TOPS_RecordDeleted          = (1<<2),
   TOPS_RecordModified         = (1<<1),
   TOPS_RecordNew              = (1<<0)
}TOPS_Status;

// Definition of a Record Id. The record id value is dependent on the
// table type.  For example, the record id for the Application Table is an
// Application Id, the record id for Table Info Table is a Table Id
typedef DWORD TOPS_RecordId;



// FUNCTION PROTOTYPES:


// TOPS_BeginSession is the function that is used to start the communication
// session.  This function does the initialization of the API.
// If this function is not called, or it completes with a
// FALSE value, all subsequent calls to the API will return FALSE.
//
// Input Parameters:
//   p_port - This parameter tells the software what logical com port to
```

```
//   use for communication to the handheld if p_device is a COM device.
//   Typically this is COM4.  This
//   value depends on what logical com port the computer has setup for
//   IRDA communications.  This value is NOT the same as the actual physical
//   com port used for the IRDA connection. If p_device does not specify
//   a COM device then this parameter is ignored.
//
//   p_device - This parameter tells the software what the physical device
//   type is for the connection.  For example, it could be an IRDA device,
//   or it could be a COM port device, or a MODEM, etc.  The include
//   file devices.h has the values defined for this input parameter.
//
//   p_id_string_ptr - This parameter points to a block of memory that will
//   receive the PDA-100 indentification character string returned by the
//   handheld.  This string will be null terminated and
//   contains version information that should be examined to ensure that it is
//   compatible with the caller.
//   The buffer this parameter points at must be large enough to contain the
//   id and the null terminator.
//

// Return Values:
//   The function returns TRUE if all initialization succeeded, and the
//   software successfully initiated a connect sequence with the handheld.
//   If this software returns TRUE, it indicates that the API has
//   established a connection with the handheld and that subsequent API
//   function calls should successfully execute.  If this function returns
//   FALSE it indicates that a connection could not be established with
//   the handheld.  The I/O API Error Buffer can be examined to
//   retrieve additional error information.
//
//
// Usage Notes:
//   1.  This API was developed to allow a sesssion to be initiated from both
//       the PC and from the handheld.
//       TOPS_BeginSession will create the connection to the handheld. This
//       will cause automatic power-on of the handheld if it is connected via
//       the wire.  Since the handheld can also initiate a session, the PC
//       software must poll the handheld in order to determine when the
//       handheld is initiating a session. TOPS_BeginSession can not be used
//       to poll the handheld because each time it is called it would turn on
//       the handheld, this would waste the handheld battery and look strange
//       with the handheld going on and off.  Because of this, the function
//       TOPS_CheckComm should be used to poll for a handheld initiated session.
//       TOPS_CheckComm will not turn on the handheld because it uses the IRDA
//       sniff protocol to detect the presence of the handheld. See
//       TOPS_CheckComm for more details on it's usage. If the session is
//       initiated from the PC, TOPS_CheckComm shouldn't be called, and
//       TOPS_BeginSession can be called directly.

TOPS_ReturnValues DLLEXPORT TOPS_BeginSession(int p_port,
                                              IO_DeviceIdType p_device,
                                              char *p_id_string);
```

```
// TOPS_EndSession is the function that does the shutdown of the
// TOPS API.  This includes doing the OBEX disconnect sequence for the PDA-100.
//
// This must be the last API routine that is called.
//
//
// Input Parameters:  NONE
//
//
// Return Values:
//   The function returns TRUE if the disconnect packet sent to the handheld
//   was transmitted successfully.
//   The function returns FALSE if the disconnect packet sent to the handheld
//   was unsuccessful or a disconnect was attempted when not connected.  The
//   I/O API Error Buffer can be examined to retrieve additional error
//   information.
//   Regardless of whether this function returns TRUE or FALSE, the IO API will
//   be shutdown.
//

TOPS_ReturnValues DLLEXPORT TOPS_EndSession(void);




// TOPS_GetTags is the function that is used to get the tags of the
// records of a table. A tag is simply a two word data structure that
// contains the record id and the status of the record. The status of the
// record tells if the record is new, modified, deleted, and in some tables
// whether or not it is an exclusive access record.
// Since the records of Table Info Table
// contain the statuses of all the other tables, doing a GetTags of the
// Table Info Table is a quick way to get the statuses of all the tables.
// TOPS_BeginSession must be called before this function will properly
// execute.
//
// Input Parameters:
//   p_table_type - This parameter tells the software which table to get
//   the tags of all the records from.
//   p_selector - This parameter allows the user to further specify which
//   tags to return.  This parameter allows the caller to get:
//     1. all the tags,
//     2. get just the tags of modified records,
//     3. just the tags of new records,
//     4. just the tags of deleted records.
//     5. any combination of the above.
//   p_num_returned_tags - This parameter points at a memory location that will
//   receive the actual number of tags that were returned by the handheld.
//   p_returned_tags_ptr - This parameter points at a buffer that will
//   receive the returned tags.
//
// Return Values:
```

```
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.
//


// Definition of the tag list paramter.
typedef struct TOPS_TagDef
{
   TOPS_RecordId Id;
   WORD          Reserved;
   WORD          Status;    // can't make type TOPS_Status because enum is
                            // type int which is 4 bytes
}TOPS_TagType;

typedef TOPS_TagType TOPS_TagList[]; // an array of tags

// Definition of the Selector parameter.
typedef enum TOPS_Selector
{
   TOPS_GetPrivate             = (1<<8),
   TOPS_GetAll                 = (1<<3),
   TOPS_GetDeleted             = (1<<2),
   TOPS_GetModified            = (1<<1),
   TOPS_GetNew                 = (1<<0)
}TOPS_Selector;

TOPS_ReturnValues DLLEXPORT TOPS_GetTags(TOPS_TableType p_table_type,
                                         TOPS_Selector  p_selector,
                                         unsigned int  *p_num_of_returned_tags,
                                         TOPS_TagList  *p_returned_tags_ptr);



// TOPS_PutTags is the function that is used to set the statuses of the
// records of a table. A tag is simply a two word data structure that
// contains the record id and the status of the record. The status of the
// record tells if the record is new, modified, deleted, and in some tables
// whether or not it is an exclusive access record.
// Since the records of Table Info Table
// contain the statuses of all the other tables, doing a PutTags of the
// Table Info Table is a quick way to delete/create one or more tables.
// TOPS_BeginSession must be called before this function will properly
// execute.
//
// Input Parameters:
//   p_table_type - This parameter identifies which table will have its
//   record tags changed.
//   p_num_of_tags - This parameter identifies the number of tags to change.
//   This parameter must be equal to the number of tags pointed to by the
//   input parameter p_tags_ptr.
//   p_tags_ptr - This parameter points at a list of the new tags for
```

```
//   the specifed records and table.  In the list, there is one tag per
//   record to modify.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.
//

TOPS_ReturnValues DLLEXPORT TOPS_PutTags(TOPS_TableType p_table_type,
                                         WORD           p_num_tags,
                                         TOPS_TagList  *p_tags_ptr);




// TOPS_GetRecord is the function that is used to get a record from a table.
//
// Input Parameters:
//   p_table_type - This parameter tells the software which table to get
//   the record from.
//   p_record_id - The record id to get.
//   p_returned_rec_status_ptr - This parameter points at a memory location that
//   will receive the status of the returned record.
//   p_returned_rec_ptr - This parameter points to the record structure that
//   will be filled with the returned record.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.

TOPS_ReturnValues DLLEXPORT TOPS_GetRecord( TOPS_TableType p_table_type,
                                         TOPS_RecordId  p_record_id,
                                         WORD          *p_returned_rec_status_ptr,
                                         TOPS_Record   *p_returned_rec_ptr);



// TOPS_PutRecord is the function that is used to put a record to a table.
//
// Input Parameters:
//   p_table_type - This parameter tells the software which table to put
//   the record to.
//   p_record_id - The record id to put.
//   p_record_status - The status of the put record.
//   p_record_ptr - Pointer to the record to put.
//
// Return Values:
//   This function can return 0, 1, or 2.
```

```
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.

TOPS_ReturnValues DLLEXPORT TOPS_PutRecord( TOPS_TableType p_table_type,
                                            TOPS_RecordId  p_record_id,
                                            WORD           p_record_status,
                                            TOPS_Record   *p_record_ptr );


// TOPS_GetUnexpectedResponse is the function that is used to retrieve the
// unexpected response value from the handheld.  If one of the TOPS API
// functions return the unexpected response value (2), then the user
// should call this function to get the unexpected response data.
//
// Input Parameters:
//   p_error_category_ptr - Pointer to the location to receive the error
//   category from the handheld.
//   p_error_id_ptr - Pointer to the locationt to receive the error id from
//   the handheld.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.

// Definition of return parameter values
typedef enum TOPS_ErrorCategoryDef
{
 CommonError          = 0,
 RequestSpecificError = 1   // currently no request specific errors identified
} TOPS_ErrorCategory;

typedef enum TOPS_CommonErrorDef
{
  Password_Required   = 0,
  Object_Not_Found    = 1,
  Insufficient_Memory = 2,
  Invalid_OBEX_Opcode = 3,
  Id_Unavailable      = 4,
  Bad_Usage           = 5,
  Bad_Object_Size     = 6
}TOPS_ErrorId;


TOPS_ReturnValues DLLEXPORT TOPS_GetUnexpectedResponse(
                                    TOPS_ErrorCategory   *p_error_category_ptr,
                                    TOPS_ErrorId         *p_error_id_ptr);


 // Sys Info

typedef enum TOPS_SysInfoItemsDef
```

```
{
  // Category A Read/Write Item 0
  TOPS_DatabaseStatusItem=0,
  // Category A Read/Write, Item 1
  TOPS_PC_Storage_A_Item=1,
  // Category A Read/Write, Item 2
  TOPS_HH_ClockItem=2,
  // Category A Read/Write, Item 3
  TOPS_CityTimeSettingsItem=3,
  // Category A Read/Write, Item 4
  TOPS_DailyAlarmItem=4,
  // Category A Read/Write, Item 5
  TOPS_SystemSettingsItem=5,
  // Category A Read/Write, Item 6
  TOPS_DisplayFormatItem=6,
  // Category A Read/Write, Item 7
  TOPS_OwnerInfoItem=7,
  // Category A Read/Write, Item 8
  TOPS_OptimizationSettingsItem=8,
  // Category A Read/Write, Item 9
  TOPS_PrivateLabelItem=9,  // text for "private" checkbox
  // Category A Read/Write, Item 10
  TOPS_CalendarAppPreferencesItem=10,
  // Category A Read/Write, Item 11
  TOPS_TaskAppPreferencesItem=11,
  // Category A Read/Write, Item 12
  TOPS_AddressAppPreferencesItem=12,
  // Category A Read/Write Item 13
  TOPS_JustTypeDictionaryStatusItem=13,
  // Category B Read/Write, Item 0
  TOPS_HandheldIdItem=14,
  // Category B Read/Write, Item 1
  TOPS_PC_Storage_B_Item=15,
  // Category B Read/Write, Item 2
  TOPS_TZandDSTDatabaseEntryItem=16,
  // Category B Read/Write, Item 3
  TOPS_SyncMessageItem=17, // message to display during synchronization
  // Category B Read/Write, Item 4
  TOPS_SplashScreenItem=18,
  // Category C Read/Write, Item 0
  TOPS_PasswordItem=19,
  //*** Read-Only Items:
  // Category A Read-Only, Item 0
  TOPS_ModelInfoItem=20,
  // Category A Read-Only, Item 1
  TOPS_MemoryCapacityItem=21,
  // Category A Read-Only, Item 2
  TOPS_MemoryAvailableItem=22,
  // Category A Read-Only, Item 3
  TOPS_LocalTZandDSTItem=23,
  // Category A Read-Only, Item 4
  TOPS_City1to4TZandDSTItem=24,
  // Category A Read-Only, Item 5
```

```
  TOPS_JustTypeDictionarySizeItem=25,
  // Category A Read-Only, Item 6
  TOPS_LanguageBundleItem=26,

  // All Category A Items
  TOPS_GroupedItem=27,
  // Category B Read-Only Item 0
  TOPS_UserRequestedSyncItem=28,
  // Category B Read-Only Item 1
  TOPS_UserCancelledSyncItem=29,
  // Category D Read-Only Item 0
  TOPS_NewRecordIdItem=30,
  // Category D Read-Only Item 1
  TOPS_NewApplicationIdItem=31,
  TOPS_MaxSysInfoItem
}TOPS_SysInfoItems;


 // Read/write structures
 //   Theses are the structures that can be both written and read by the
 //   PC software.

 // Category A Read/Write

 // Category A Read/Write, Item 0
typedef BYTE TOPS_DatabaseStatus[16];

// Category A Read/Write, Item 1
typedef BYTE TOPS_PC_Storage_A[64];

typedef struct TOPS_HH_ClockDataAndTimeDef
{ // Category A Read/Write, Item 2
  TOPS_Date Date;
  TOPS_Time Time;
}TOPS_HH_Clock;
typedef struct TOPS_CityTimeSettings
{ // Category A Read/Write, Item 3
  WORD LocalIndex;
  WORD City1Index;
  WORD City2Index;
  WORD City3Index;
  WORD City4Index;
}TOPS_CityTimeSettings;
typedef struct TOPS_DailyAlarmDef
{ // Category A Read/Write, Item 4
  BYTE Hour; // (0..23)
  BYTE Minute; // (0..59)
  BYTE On_Off; // 0=off, 1=on
}TOPS_DailyAlarm;
typedef struct TOPS_SystemSettingsDef
{ // Category A Read/Write, Item 5
  BYTE PowerDownMinutes; // (1,2, or 3 minutes)
  BYTE BacklightOnTimer; // (0=5 seconds, 1=10s, 2=15s)
```

```
  BYTE AlarmSound;       // (0=no, 1=yes)
  BYTE PenTone;          // (0=no, 1=yes)
  BYTE StartupScreen;    // (0=don't show, 1=show default, 2=show graphic)
}TOPS_SystemSettings;

typedef struct TOPS_DisplayFormatDef
{ // Category A Read/Write, Item 6
  BYTE DateDisplay;  // 0=M/D/Y, 1=D/M/Y
  BYTE TimeDisplay;  // 0=12hr,  1=24hr
  BYTE NumberDisplay; // 0=1,234.56   1=1.234,56
  BYTE StartOfWeek;   // 0=Sunday,  1=Monday
}TOPS_DisplayFormat;

typedef struct TOPS_OwnerInfoDef
{ // Category A Read/Write, Item 7
  BYTE Name[32];
  BYTE Note[160];
}TOPS_OwnerInfo;

typedef struct TOPS_OptimizationSettingsDef
{ // Category A Read/Write, Item 8
  BYTE Frequency;          //0=manually, 1=daily, 2=weekly
  BYTE HourToOptimize;     // (0..23)
  BYTE DayToOptimize;      // (0..6) 0 = Sunday
}TOPS_OptimizationSettings;

// Category A Read/Write, Item 9
typedef BYTE TOPS_PrivateLabel[16];  // text for "private" checkbox

typedef struct TOPS_CalendarAppPreferencesDef
{ // Category A Read/Write, Item 10
  BYTE StartDayAt;      // (0..18)
  BYTE EventTimeStyle;  // 0=Start time/end time, 1=Start time and duration
  BYTE DefaultDuration; // 0=15min, 1=30min, 2=1hour
  BYTE TimeSlot;        // 0=15min, 1=30min, 2=1hour
  BYTE AdvanceNotice;   // 0=0min,1=5,2=10,3=15,4=30,5=45,6=1hour,7=1.5,8=2hours
}TOPS_CalendarAppPreferences;

typedef struct TOPS_TaskAppPreferencesDef
{ // Category A Read/Write, Item 11
  BYTE MaxAlphaPriority;   //1=No Numeric Priority, 2=B,3=C,4=D,5=E
  BYTE MaxNumericPriority; //1=No Numeric Priority, 2=2,3=3.....9=9
  BYTE ShowPriorityInList; // 0=no, 1=yes
  BYTE ShowDueDateInList;  // 0=no, 1=yes
}TOPS_TaskAppPreferences;

typedef struct TOPS_AddressAppPreferencesDef
{  // Category A Read/Write, Item 12
  BYTE AddressFormat; // 0=NorthAmerican, 1=European
}TOPS_AddressAppPreferences;

typedef struct TOPS_JustTypeDictionaryStatusDef
{  // Category A Read/Write, Item 13
```

```
  WORD NewWords;
  BYTE CorruptionFlag; // 0=Good, 1=Corrupt
}TOPS_JustTypeDictionaryStatus;

// Category B Read/Write

// Category B Read/Write, Item 0
typedef BYTE TOPS_HandheldId[16];

// Category B Read/Write, Item 1
typedef BYTE TOPS_PC_Storage_B[64];



// Special info for daylight savings time data:
// 1. When month is 0, the following two bytes stands for day of year
//   (in lo-byte, hi-byte format). Leap day is counted in calculation
//   for day of year.
// 2. When month is not 0 and week is 0 then month and date stand for normal
//    calendar day.
// 3. When month <> 0 and week <> 0  then this specifies day d of week w of
//    month m. The day d must be between 1 (Sunday) and 7. The week w must
//    be between 1 and 5; week 1 is the first week in which day d occurs,
//    and week 5 specifies the last d day in the month.

typedef struct TOPS_DSTDateDef
{
  BYTE Month;
  BYTE Week;
  BYTE Day;
}TOPS_DSTDate;

typedef struct TOPS_DSTTimeDef
{
  BYTE Hour;    // (0..23)
  BYTE Minutes; // (0..59)
}TOPS_DSTTime;

typedef struct TOPS_TZandDSTDef
{
  BYTE HourOffset;   // Timezone hour offset from GMT (0..12)
  BYTE MinuteOffset; // Timezone minute offset from GMT (0..59)
  BYTE DSTChange;    // DST change in minutes (-60..+60)
  TOPS_DSTDate StartDate;
  TOPS_DSTTime StartTime;
  TOPS_DSTDate EndDate;
  TOPS_DSTTime EndTime;
  BYTE PC_Storage[2];
  BYTE CityName[20];
}TOPS_TZandDST;

// Category B Read/Write, Item 2
typedef struct TOPS_TZandDSTDatabaseEntryDef
```

```
{
  TOPS_TZandDST Entries[100];
}TOPS_TZandDSTDatabaseEntry;

// Category B Read/Write, Item 3
typedef struct TOPS_SyncMessageDef
{
  BYTE SyncMessage[64];        // message to display during synchronization
  BYTE EndOfSyncMessage[64];   // message displayed after synchronization
  BYTE SyncResultsMessage[64]; // message describing result of sync
}TOPS_SyncMessage;


// The image is a sequence of rows. The first row has the top line of the image.
// Each row has just enough bytes to hold the pixels for the width of an image.
// Any remaining unused bits are filled with 0. The number of rows is equal to
// the height of the image in pixels.
// if you plan to display the owner info over the splash screen at startup,
// then choose or create an image that will still look good if the lower 1/3rd
// of the screen is covered.

// Category B Read/Write, Item 4
typedef struct TOPS_SplashScreenDef
{
  BYTE Image[4800]; // image
}TOPS_SplashScreen;

// Category C Read/Write, Item 0
typedef BYTE TOPS_Password[8];









//*********************************************************************
// Read only structures.
//   The following structures can only be read by the PC software
//*********************************************************************
// Category A Read-Only

// Category A Read-Only, Item 0
typedef char TOPS_ModelInfo[sizeof("PDA100 VM.m.P.p")-1];
// M : major version #
// m : minor version #
// P :  major patch #
// p :  minor patch #
// For example, each of the units in the first shipment would have Model Info
// of  "PDA100 V1.0.0.0". If they received patch version 2.4, then it would
// be "PDA100 V1.0.2.4"
```

```
// Category A Read-Only, Item 1
typedef struct TOPS_MemoryCapacityDef
{
  DWORD TOPS_ROMSize;
  DWORD TOPS_SRAMSize;
  DWORD TOPS_FlashSize;
}TOPS_MemoryCapacity;
// Category A Read-Only, Item 2
typedef struct TOPS_MemoryAvailableDef
{
  DWORD TOPS_BytesAvailable;
  DWORD TOPS_BytesDeleted;
  DWORD TOPS_BytesActive;
}TOPS_MemoryAvailable;
// Category A Read-Only, Item 3
typedef TOPS_TZandDST TOPS_LocalTZandDST;
// Category A Read-Only, Item 4
typedef TOPS_TZandDST TOPS_City1to4TZandDST[4];
// Category A Read-Only, Item 5
typedef WORD TOPS_JustTypeDictionarySize;
// Category A Read-Only, Item 6
typedef TOPS_Byte TOPS_LanguageBundleSysInfo; // 0=none loaded,
1=English,2=French,3=German,4=Spanish

// Category B Read-Only

typedef BYTE TOPS_UserRequestedSync;
typedef BYTE TOPS_UserCancelledSync;

// Category D Read-Only
typedef DWORD TOPS_NewRecordId;
typedef WORD TOPS_NewApplicationId;

// Category D Items also require parameters on the call to TOPS_GetSysInfo.

// Parameter definition for getting new record id
typedef struct TOPS_NewRecIdParmDef
{
  WORD RecordSize;  // size of the new record
  WORD TableId;     // table of the new record
}TOPS_NewRecIdParm;
// Parameter definition for getting new application id
typedef struct TOPS_NewAppIdParmDef
{
  WORD TOPS_NewAppSize;  // size of new application
}TOPS_NewAppIdParm;

typedef union TOPS_SysInfoParmsDef
{
  TOPS_NewRecIdParm RecordIdParm;
  TOPS_NewAppIdParm AppIdParm;
}TOPS_SysInfoParms;
```

```
// Category A Grouped Object (Read Only)
typedef struct TOPS_GroupedObjectDef
{
  // Category A Read/Write Item 0
  TOPS_DatabaseStatus DatabaseStatus;
  // Category A Read/Write, Item 1
  TOPS_PC_Storage_A    PC_Storage;
  // Category A Read/Write, Item 2
  TOPS_HH_Clock HH_Clock;
  // Category A Read/Write, Item 3
  TOPS_CityTimeSettings CityTimeSettings;
  // Category A Read/Write, Item 4
  TOPS_DailyAlarm Alarm;
  // Category A Read/Write, Item 5
  TOPS_SystemSettings SystemSettings;
  // Category A Read/Write, Item 6
  TOPS_DisplayFormat DisplayFormat;
  // Category A Read/Write, Item 7
  TOPS_OwnerInfo OwnerInfo;
  // Category A Read/Write, Item 8
  TOPS_OptimizationSettings OptimizationSettings;
  // Category A Read/Write, Item 9
  TOPS_PrivateLabel PrivateLabel;  // text for "private" checkbox
  // Category A Read/Write, Item 10
  TOPS_CalendarAppPreferences CalendarAppPreferences;
  // Category A Read/Write, Item 11
  TOPS_TaskAppPreferences TaskPreferences;
  // Category A Read/Write, Item 12
  TOPS_AddressAppPreferences AddressAppPreferences;
  // Category A Read/Write, Item 13
  TOPS_JustTypeDictionaryStatus JustTypeStatus;
  // Category A Read-Only, Item 0
  TOPS_ModelInfo ModelInfo;
  // Category A Read-Only, Item 1
  TOPS_MemoryCapacity MemoryCapacity;
  // Category A Read-Only, Item 2
  TOPS_MemoryAvailable MemoryAvailable;
  // Category A Read-Only, Item 3
  TOPS_LocalTZandDST TOPS_LocalTZandDST;
  // Category A Read-Only, Item 4
  TOPS_City1to4TZandDST City1to4TZandDST;
  // Category A Read-Only, Item 5
  TOPS_JustTypeDictionarySize JustTypeSize;
  // Category A Read-Only, Item 6
  TOPS_LanguageBundleSysInfo LanguageBundle;
}TOPS_GroupedObject;


typedef union TOPS_ReadWriteSysInfoDef
{
  // Category A Read/Write Item 0
  TOPS_DatabaseStatus DatabaseStatus;
```

```
  // Category A Read/Write, Item 1
  TOPS_PC_Storage_A  PC_StorageA;
  // Category A Read/Write, Item 2
  TOPS_HH_Clock HH_Clock;
  // Category A Read/Write, Item 3
  TOPS_CityTimeSettings CityTimeSettings;
  // Category A Read/Write, Item 4
  TOPS_DailyAlarm DailyAlarm;
  // Category A Read/Write, Item 5
  TOPS_SystemSettings SystemSettings;
  // Category A Read/Write, Item 6
  TOPS_DisplayFormat DisplayFormat;
  // Category A Read/Write, Item 7
  TOPS_OwnerInfo OwnerInfo;
  // Category A Read/Write, Item 8
  TOPS_OptimizationSettings OptimizationSettings;
  // Category A Read/Write, Item 9
  TOPS_PrivateLabel PrivateLabel;  // text for "private" checkbox
  // Category A Read/Write, Item 10
  TOPS_CalendarAppPreferences CalendarAppPreferences;
  // Category A Read/Write, Item 11
  TOPS_TaskAppPreferences TaskAppPreferences;
  // Category A Read/Write, Item 12
  TOPS_AddressAppPreferences AddressAppPreferences;
  // Category A Read/Write, Item 13
  TOPS_JustTypeDictionaryStatus JustTypeStatus;
  // Category B Read/Write, Item 0
  TOPS_HandheldId HandheldId;
  // Category B Read/Write, Item 1
  TOPS_PC_Storage_B PC_StorageB;
  // Category B Read/Write, Item 2
  TOPS_TZandDSTDatabaseEntry TZandDSTDatabaseEntry;
  // Category B Read/Write, Item 3
  TOPS_SyncMessage SyncMessage; // message to display during synchronization
  // Category B Read/Write, Item 4
  TOPS_SplashScreen SplashScreen;
  // Category C Read/Write, Item 0
  TOPS_Password Password;
}TOPS_ReadWriteSysInfo;

typedef union TOPS_ReadOnlyDef
{
  // Category A Read-Only, Item 0
  TOPS_ModelInfo ModelInfo;
  // Category A Read-Only, Item 1
  TOPS_MemoryCapacity MemoryCapacity;
  // Category A Read-Only, Item 2
  TOPS_MemoryAvailable MemoryAvailable;
  // Category A Read-Only, Item 3
  TOPS_LocalTZandDST LocalTZandDST;
  // Category A Read-Only, Item 4
  TOPS_City1to4TZandDST City1to4TZandDST;
  // Category A Read-Only, Item 5
```

```
  TOPS_JustTypeDictionarySize JustTypeSize;
  // Category A Read-Only, Item 6
  TOPS_LanguageBundleSysInfo LanguageBundle;
  // All Category A Items
  TOPS_GroupedObject GroupedObject;
  // Category B Read-Only Item 0
  TOPS_UserRequestedSync UserRequestedSync;
  // Category B Read-Only Item 1
  TOPS_UserCancelledSync UserCancelledSync;
  // Category D Read-Only Item 0
  TOPS_NewRecordId NewRecordId;
  // Category D Read-Only Item 1
  TOPS_NewApplicationId NewApplicationId;
}TOPS_ReadOnlySysInfo;

typedef union TOPS_SysInfoDef
{
  TOPS_ReadWriteSysInfo ReadWrite;
  TOPS_ReadOnlySysInfo  ReadOnly;
}TOPS_SysInfo;


// TOPS_PutSysInfo is the function that is used to put read/write SysInfo Items
//
// Input Parameters:
//   p_item_type - This parameter identifies what type item to put.
//   p_item_ptr - Pointer to the item to put.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.

TOPS_ReturnValues DLLEXPORT TOPS_PutSysInfo( TOPS_SysInfoItems      p_item_type,
                                             TOPS_ReadWriteSysInfo *p_item_ptr );

// TOPS_GetSysInfo is the function that is used to get SysInfo Items
//
// Input Parameters:
//   p_item_type - This parameter identifies what type item to get.
//   p_parameters - This is the pointer to the parameter(s) for the get.
//   p_item_ptr - Pointer to memory location to receive the SysInfo.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.

TOPS_ReturnValues DLLEXPORT TOPS_GetSysInfo( TOPS_SysInfoItems  p_item_type,
                                             TOPS_SysInfoParms *p_parameters,
                                             TOPS_SysInfo      *p_item_ptr );
```

```
// TOPS_PutCommand is the function that is used to send commands
//
// Input Parameters:
//   p_item_type - This parameter identifies the command.
//   p_parm_ptr - This is the pointer to the parameter(s) to send. See the
//                TOPS document for a description of the parameters.
//
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.

typedef enum TOPS_CommandTypeDef
{
  TOPS_UnlockWithPassword   = 0,
  TOPS_CompactMemory        = 1,
  TOPS_SyncMessages         = 2,
  TOPS_DeleteUserDictionary = 3,
  TOPS_MaxCommand
}TOPS_CommandType;

#define TOPS_UnlockWithPasswordParameterSize   8
#define TOPS_CompactMemoryParameterSize        0
#define TOPS_SyncMessagesParameterSize         64*3
#define TOPS_DeleteUserDictionaryParameterSize 0
#define TOPS_MaxCmdParameterSize               64*3
typedef BYTE TOPS_CommandParameter[TOPS_MaxCmdParameterSize];

TOPS_ReturnValues DLLEXPORT TOPS_PutCommand(TOPS_CommandType       p_item_type,
                                            TOPS_CommandParameter *p_parm_ptr );

// TOPS_SetupCallback is the function that is used to setup a routine that will
// be called by the API when it is in a time consuming loop.  The callback
// routine must return TRUE or FALSE.  If the callback returns FALSE the API
// will stop its current execution and return.
//
// Input Parameters:
//   p_IO_Callbackfunction - Pointer to the callback function.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.

TOPS_ReturnValues DLLEXPORT TOPS_SetupCallback(int (_USERENTRY
*p_IO_Callbackfunction)(void));

// TOPS_CancelCallback is the function that is used to remove a callback that
// was setup using TOPS_SetupCallback.
//
```

```
// Input Parameters:
//   None.
//
// Return Values:
//   This function can return 0, 1, or 2.
//   0 indicates a failure.
//   1 indicates the function executed successfully.
//   2 indicates an unexpected response from the handheld.



TOPS_ReturnValues DLLEXPORT TOPS_CancelCallback(void);


// TOPS_SetupDisconnectWarningCallback is the function that is used to setup
// a routine that will be called by the API when the IRLAP layer of the TI
// stack has not received any response from the handheld. This could be due
// to the handheld being removed from the cradle, or the cable removed, etc.
// The API calls the callback with a parameter indicating the current
// disconnect count.  IRLAP sends the first Warning when the count reaches
// 3, therefore the first number received will be 3.  If the handheld starts
// responding, the callback will be called with a value of 0xFFFF.  If the
// handheld never returns, the IRLAP layer will timeout after the IRLAP
// negotiatied timeout, and this function will return 0xFFFE.   This
// function is intended to allow the API user to pop up a message to the end
// product user when an obstruction in the data path occurs. The API user knows
// that the message can be removed when it receives the 0xFFFF or 0xFFFE value.
// Data path obstruction in wired communication this is not as probable as in
// a true IR usage.
//
// Input Parameters:
//   p_IO_Callbackfunction - Pointer to the callback function.
//
// Return Values:
//   NONE
//
// Callback Input Parameters:
//   integer that will get the current disconnect count

void DLLEXPORT TOPS_SetupDisconnectWarningCallback(void (_USERENTRY
*p_IO_Callbackfunction)(int));

// TOPS_CancelDisconnectWarningCallback is the function that is used to remove
// a callback that was setup using TOPS_SetupDisconnectWarningCallback.
//
// Input Parameters:
//   None.
//
// Return Values:
//   None.

void DLLEXPORT TOPS_CancelDisconnectWarningCallback(void);
```

```
// TOPS_CheckComm is the function that is used to determine if there is
// a handheld ready to communicate. This routine should be called
// repeatedly until it returns TRUE, and then TOPS_BeginSession should be
// called with the same values for the port and device.
//
// Input Parameters:
//   p_port - This parameter tells the software what com port to
//   use for checking for communication to the handheld. Typically if the
//   Win95 stack is to be used (for IR) this value is COM4, if the TI stack
//   is to be used (for Wired Communication) this value is COM1 or COM2.
//   p_device - Identifies which connection to check, e.g. wired or IRDA.
//   This parameter essentially dictates which stack is going to be used and
//   the method by which a connection will be checked. There are 2 major
//   choices for p_device, IO_ComPort, and IO_IRDA:
//     When the input parameter is IO_ComPort it specifies communication
//   through the com port (ie the TI stack is used), then this function
//   initiates the low power IrDA sniffing algorithms.  Subsequent calls to
//   this function then check to see if a sniff detection has occurred. If a
//   detection  has occurred then this function returns TOPS_CheckCommSuccess,
//   and the user should then call TOPS_BeginSession to start the handheld
//   initiated session.  Note that this function only returns
//   TOPS_CheckCommSuccess when the handheld has attempted
//   to initiate a session, it is OK to call TOPS_BeginSession at anytime if
//   the PC session is to be initiated by the PC.  I.E. you don't have to wait
//   for this function to return TOPS_CheckCommSuccess before initiating a
//   session from the PC.
//     When the input paramter is IO_IRDA it specifies communication through
//   the IR Port, in this case the Win95 stack is intended to be used.  The
//   function will attempt to connect and if the connect attempt fails then
//   the function returns TOPS_CheckCommFindFailure. If it succeeds the
//   function returns TOPS_CheckCommSuccess, and the user should then call
//   TOPS_BeginSession.  Note that the Win95 stack has no way to initiate
//   a sniff, this means that when this function returns TOPS_CheckCommSuccess
//   the handheld has been connected to and can be communicated with regardless
//   of whether or not the handheld user has initiated a connection on the
//   handheld unit.
//
// Return Values:
//   0 - No connection available, call this function again at a later time.
//   1 - A connection is available, so it is safe to call TOPS_BeginSession.
//   2 - A true error has occurred, most likely the port is already in use.
//
// Usage Notes:
//   1.  There is nothing that prevents the user from using the TI Stack for
//       communication over the IR port rather than the serial port, (or for
//       using the Win95 stack for the serial port rather than the IR port).
//       However the original intent of the TI stack was to provide a
//       stand-alone stack for the wired port such that the Win95 stack and
//       the TI Stack could function simultaneosly on different ports.

// Function values returned by TOPS_CheckComm.
typedef enum TOPS_CheckCommReturnValuesDef
```

```
{
   TOPS_CheckCommFindFailure  = 0,
   TOPS_CheckCommSuccess      = 1,
   TOPS_CheckCommError        = 2
}TOPS_CheckCommReturnValues;

TOPS_CheckCommReturnValues DLLEXPORT TOPS_CheckComm(BYTE p_port,
                                                    IO_DeviceIdType p_device);

// restore previous alignment
#ifdef __BORLANDC__
#pragma option -a-.                                  // Borland restore
#else
#pragma pack( pop, alignment_before_tops_includes ) // MFC restore
#endif

#ifdef __cplusplus
}
#endif
#endif

//---------------------------------------------------------------------------
//  End of File: tops_pda.h
//---------------------------------------------------------------------------
```

io_devices.h

```
/*****************************************************************************\
*  io_devices.h
*
*  Header file that defines the device paths of the I/O API software
*
\*****************************************************************************/


// If a new device is added an entry must be added to this table.
// The new id should be placed immediately prior to the InvalidHighDevice
// Entry.
enum IO_DeviceIdDef
{
   InvalidLowDevice,
   IO_ComPort,
   IO_IRDA,
   InvalidHighDevice
};
typedef enum IO_DeviceIdDef IO_DeviceIdType;
IO_DeviceIdType IO_DeviceId;


//---------------------------------------------------------------------------
//  End of File: io_devices.h
//---------------------------------------------------------------------------
```

io_error.h

```
/*****************************************************************************\
*  io_error.h
*
*  Header file that defines the error handling of the I/O API software
*
\*****************************************************************************/

// define export options
#define DLLEXPORT __export _stdcall

// These functions provide the error reporting capability of the I/O API.
// The I/O API reports errors via a circular Error Buffer.  Entries in
// the Error Buffer consist of a character string that describes the
// error followed by an optional error id that gives additional information
// for some errors.


// prototye for error handling routines


// IO_ERROR places an error string and an error id in the error buffer.


void DLLEXPORT IO_ERROR(char *p_error_string, DWORD p_error_id);
void DLLEXPORT IO_ERROR_Suspend(void);
void DLLEXPORT IO_ERROR_Resume(void);
BOOL DLLEXPORT IO_ERROR_GetErrors(WORD p_return_buffer_size,
                                  BYTE *p_return_buffer_ptr,
                                  WORD p_reset_errors_flag,
                                  WORD *p_actual_bytes_returned);

// definition of an Error Buffer Entry

typedef struct IO_ERROR_Def
{
char  error_string[150];
DWORD error_id;
} IO_ERROR_BufferType;


// defines the number of entries in the error buffer
#define IO_ERROR_NUM_ENTRIES 20

// define the Error Buffer
extern __import IO_ERROR_BufferType IO_ERROR_Buffer[IO_ERROR_NUM_ENTRIES];


// define the current index into the error buffer.  This is the index into
// the error buffer array IO_ERROR_Buffer at which the next error will
// be stored.
extern __import WORD IO_ERROR_BufferIndex;
```

```
// define the cumulative count of all errors that have been encountered.
extern __import DWORD IO_ERROR_Count;
```

Index

—R—

—S—

—T—

—U—

—W—