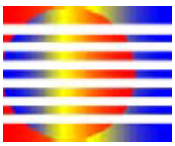


MULTISPECTRAL SOLUTIONS, INC.®

RaDeKL Radar API for Windows

Radar Developer's Kit – Lite – Application Programming Interface

Programmer's Guide

Version 2.0
Revised: 05 OCT 07

20300 Century Boulevard
Germantown, MD 20874-1132
Phone: (301) 528-1745
Fax: (301) 528-1749

Website: www.multispectral.com

NOTE: The RaDeKL radar unit has been tested to comply with FCC Part 15, Subpart C for WBT devices. Changes or modifications to the radiating elements of RaDeKL not expressly approved by the party responsible for compliance could void the user's authority to operate the equipment.

NOTE: The RaDeKL radar unit has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

—Reorient or relocate the receiving antenna.
—Increase the separation between the equipment and receiver.
—Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
—Consult the dealer or an experienced radio/TV technician for help.

Table of Contents

Part I: Welcome to the RaDeKL Radar API

Introduction

The RaDeKL Radar API provides a programming interface to the Multispectral Solution Inc. (MSSI) RaDeKL Radar device. Using this interface removes the burden of designing code to access the USB interface and dealing with the specifics of radar commands and operational parameter settings. Instead, this interface provides a set of RaDeKL Radar specific functions to list available devices, open/close a specific radar device, set operational parameters and request single and continuous range detection data.
Please refer to the enclosed folder "*Code Example"* for a complete and functional code example (in C) on how to list devices, open a device, set operational parameters, read range detection data and close the device.

USB Driver Installation

The RaDeKL Radar USB port is based on the FTDI FT2232C USB Chip, which requires a driver to operate.

> ***Note:*** Do not download any updated drivers directly from the FTDI website! The RaDeKL Radar uses a custom PID (Product ID) in the USB definition and therefore requires a custom driver. Please use only the driver provided with RaDeKL Radar devices.

Before the RaDeKL Radar device can be used, the USB driver must be installed. This driver can only be installed if a RaDeKL Radar device is physically plugged into the USB port. Windows then detects the presence of a new USB device and uses its Plug & Play feature to aid in the installation of the driver.

Please follow these steps:

1. Ensure that you are running Windows 2000, Windows XP (or newer) on the target PC and that the PC has an available USB 2.0 (not 1.1 or 1.2) port. Have an A-B USB cable ready.
2. Extract the distribution ZIP file, containing the API and driver files. Note the location of the resulting folder. It should contain a sub-folder named "*Driver*".
3. Connect the RaDeKL Radar device to the power supply and insert the power supply into a suitable electrical AC outlet (120 VAC).
4. Connect the RaDeKL Radar device to a free USB 2.0 port on your PC or laptop.
5. Windows should detect the new USB device and start the *Found New Hardware Wizard*.
6. **Windows XP:**
    a. Click "*No, not at this time*" and then "*Next*".
    b. Click "*Install from a list or specific location*" and then "*Next*".
    c. Click "*Search for the best driver in these locations*" and "*Include this location in the search*". Click "*Browse*" and navigate to the folder noted in step 2 and then down to the "*Driver*" folder. Click "*OK*" and then "*Next*".
    d. Windows XP will find the correct MSSI IUO driver and display a warning about the driver not having passed Windows Logo testing. Click "*Continue Anyway*".
    e. Windows will then detect a second USB device. Repeat the above steps for Windows XP to complete installation.

7. **Windows 2000:**
    a. Click "*Search for a suitable driver for my device*". Click "*Next*".
    b. Check the box "*Specify a location*". Click "*Next*".
    c. Click "*Browse*" and navigate to the folder noted in step 2 and then down to the "*Driver*" folder. Select the file named "*FTDIBUS.inf*" and click "*OK*" and then "*Next*".
    d. The driver will install. Windows will then detect a second USB device and the driver will install automatically.
8. ***Note*:** The additional USB device installed represents an unused internal port on the USB chip. This port is *not* used by RaDeKL Radar and can be entirely ignored. However, during driver installation, Windows will detect this unused port and insist on installing a driver for it.
9. The RaDeKL Radar is now ready for use. The driver needs to be installed only once for each PC. If it ever needs to be upgraded, use the *Windows Device Manager*, ensuring that the radar device is physically plugged in and powered up.

Programming Notes

In order to use the RaDeKL Radar API, please follow these steps (illustrated using Visual C++ 6.0):

1. Copy ***FTD2XX.h***, ***RaDeKLAPI.h*** and ***RaDeKLAPI.lib*** (from subfolder *"Library"* in the folder noted in step 2 above, "*USB Driver Installation* ") into the folder that contains the development environment of your project. Ensure that the files are *copied* (not *moved*) otherwise you won't find them for the next project.

2. Include ***RaDeKLAPI.h*** in your C program:

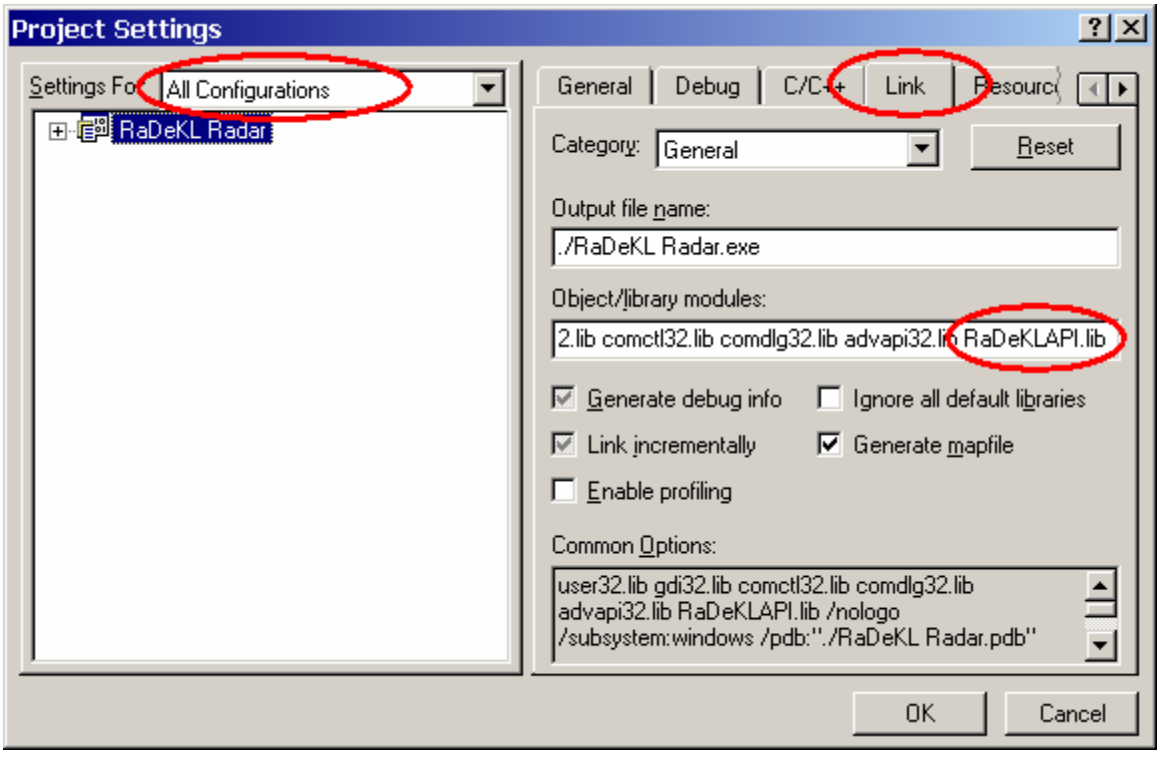
   ```
   #include "RaDeKLAPI.h"
   ```

   Note**:** *RaDeKLAPI.h* implicitly includes *FTD2XX.h*

3. Include ***RaDeKLAPI.lib*** in your project link command:

   In Visual C++ (6.0), click *Project* > *Settings*. Select "*All Configurations*" and "*Link*". Add *RaDeKLAPI.lib* to the "*Object/library modules*" as follows:

   Note**:** The *FTD2XX.lib* library is already included in *RaDeKLAPI.lib* and therefore does not need to be included in your project link, nor does it need to be present in your development environment folder. *RaDeKLAPI.lib* is entirely self-contained.



You can now use the RaDeKL API functions to operate the radar unit. The next section documents the usage of the various RaDeKL API functions available.

Usage Warnings

For proper operation of the RaDeKL radar unit using this API, please consider these warnings:

1. When operating the RaDeKL Radar in *continuous* detection mode, range data is continuously streamed from the radar unit to the PC. While in this mode, the only operation, other than reading this data, should be to stop continuous mode (using *RaDeKL_StopContinuousDetection* or a low-level register write to do the same). **Do not** issue any other register read and/or write commands (or high-level functions that read/write registers), as this will interfere with the continuous data stream presented in *continuous* detection mode. **Note:** most functions that update (write) RaDeKL registers attempt to read the data back to ensure that it was written correctly. Therefore any write-register function (RaDeKL_SetXxxxx) will result in a read after write, which will interfere with an ongoing *continuous* detection, producing unpredictable results (most likely resulting in an input/output error). If a register must be updated, please first stop *continuous* detection mode, change the register value and then restart *continuous* detection mode.
2. When operating the RaDeKL Radar in *continuous* detection mode and an error occurs (i.e. the return value of the function called, if specified, is not equal RaDeKL_OK), it is up to the caller to:
    a. Attempt to stop continuous detection mode on the radar unit by issuing a call to *RaDeKL_StopContinuousDetection*. Depending on the specific error, it might not be possible to actually communicate this command to the radar. The return code of this call should therefore be ignored in this case only.
    b. Ensure that the user interface or GUI appropriately shows that the radar is no longer operating in continuous mode. This may entail updating indicators or graying out buttons.
    c. In many error cases it may be necessary to close (if possible) and re-open the radar device in order to re-establish communications with the unit (using *RaDeKL_CloseRadar* and *RaDeKL_OpenRadar*, respectively). In some (rare) cases the unit may have to be either power-cycled or unplugged and re-connected to the USB cable. Contrary to the FTDI USB Chip documentation, this cannot always be done under program control.

Part II: Status Formatting Functions

RaDeKL_GetStatusText

Get the text associated with a status code. Most functions in this API return a numeric status code. RaDeKL_OK (0) indicates successful completion. Other codes (see below) indicate error conditions.

Format:

char ***RaDeKL_GetStatusText*** (ULONG *ftStatus*);

Parameters:

ftStatus Numeric status code as returned by most RaDeKL API functions.

Return Value:

Pointer to a null-terminated character string containing the status description.

Possible Status Codes:

Symbol	Value	Description Text
FTDI specific status codes		
RaDeKL_OK	0	Operation completed successfully
RaDeKL_INVALID_HANDLE	1	Invalid handle passed to function
RaDeKL_DEVICE_NOT_FOUND	2	Requested device not found
RaDeKL_DEVICE_NOT_OPENED	3	Specified device not open
RaDeKL_IO_ERROR	4	Input/output error
RaDeKL_INSUFFICIENT_RESOURCES	5	Insufficient resources to complete operation
RaDeKL_INVALID_PARAMETER	6	Invalid parameter passed to function
RaDeKL_INVALID_BAUD_RATE	7	Invalid baud rate specified
RaDeKL_DEVICE_NOT_OPENED_FOR_ERASE	8	Specified device not open for erase
RaDeKL_DEVICE_NOT_OPENED_FOR_WRITE	9	Specified device not open for write
RaDeKL_FAILED_TO_WRITE_DEVICE	10	Failed to write to specified device
RaDeKL_EEPROM_READ_FAILED	11	EEPROM read failed
RaDeKL_EEPROM_WRITE_FAILED	12	EEPROM write failed
RaDeKL_EEPROM_ERASE_FAILED	13	EEPROM erase failed
RaDeKL_EEPROM_NOT_PRESENT	14	EEPROM not present
RaDeKL_EEPROM_NOT_PROGRAMMED	15	EEPROM not programmed
RaDeKL_INVALID_ARGS	16	Invalid arguments passed to function
RaDeKL_NOT_SUPPORTED	17	Operation not supported
RaDeKL_OTHER_ERROR	18	USB unknown error
RaDeKL API specific status codes		
RaDeKL_READ_TIMEOUT	201	Read from radar has timed out
RaDeKL_WRITE_TIMEOUT	202	Write to radar has timed out
RaDeKL_INCORRECT_SERIAL_NUMBER	203	Serial number opened does not match request
RaDeKL_WRITE_REGISTER_FAILED	204	Register write failed
RaDeKL_READ_REGISTER_FAILED	205	Register read failed
RaDeKL_READ_DETECTION_FAILED	206	Detection data read failed
RaDeKL_BAD_THRESHOLD	207	Invalid threshold(s) specified
RaDeKL_BAD_TX_ATTEN	208	Invalid transmit attenuation specified
RaDeKL_BAD_RX_ATTEN	209	Invalid receive attenuation specified
RaDeKL_BAD_RANGE_DELAY	210	Invalid range delay specified
RaDeKL_BAD_INTERVAL	211	Invalid interval specified
RaDeKL_UNKNOWN_RESOLUTION	212	Radar has an unknown resolution

Example:

Make a RaDeKL API function call to list available devices and display an error message if the call failed.

```
ULONG   status;
DWORD   numdevs;
char    **snum = 0, **desc = 0;

status = RaDeKL_ListRadars (&numdevs, &snum, &desc);
if (status != RaDeKL_OK)
{
    printf ("Unable to list devices: %s\n", RaDeKL_GetStatusText (status));
    return status;
}

// All OK
return RaDeKL_OK;
```

RaDeKL_GetAPIVersion

Retrieve the version number of this API. The major version number is in the high-order word and the minor version number is in the low-order word. See example for details of usage.

> ***NOTE:*** The Version information returned by this function is the version of this API, **not** the version of any radar unit attached. To obtain radar unit version information, use function *RaDeKL_GetDeviceInfo* after the radar has been opened.

Format:

ULONG **RaDeKL_GetAPIVersion** ();

Parameters:

<none>

Return Value:

ULONG (32 bits) with the major version number in the high-order WORD (16 bits) and the minor version number in the low-order WORD (16 bits).

Example:

Get and display the major and minor version numbers of this API.

```
ULONG   version;
WORD    major_version;
WORD    minor_version;

version = RaDeKL_GetAPIVersion ();

major_version = (WORD)((version >> 16) & 0xFFFF);
minor_version = (WORD)(version & 0xFFFF);
printf ("This API is version %d.%d\n", major_version, minor_version);
```

Part III: Radar Management Functions

RaDeKL_ListRadars

Get a list of all RaDeKL Radar devices connected to the system. ***Note:*** This function will only retrieve RaDeKL Radar devices and will ignore other devices that may be based on the same FTDI USB Chip.

Each device has a serial number (up to 15 chars) and a device description (up to 63 chars). The serial number can then be used in a call to *RaDeKL_OpenRadar* to open the device. The serial number is unique to each unit sold, whereas the description string is unique to a particular version (or model) of RaDeKL Radars and may change with future versions.

Format:

ULONG **RaDeKL_ListRadars** (DWORD **numdevs*, char ****serial_numbers*, char ****descriptions*);

Parameters:

numdevs	Pointer to a DWORD to receive the device count. This may be NULL, in which case no device count is returned.
serial_numbers	Pointer to a pointer to a (null-terminated) list of character strings to receive the serial numbers of the connected devices. (**numdevs*) should initially be NULL, as the function allocates the list and the individual strings. See example for clarification. If NULL is passed as *numdev*, no serial numbers are returned.
descriptions	Pointer to a pointer to a (null-terminated) list of character strings to receive the descriptions of the connected devices. (**descriptions*) should initially be NULL, as the function allocates the list and the individual strings. See example for clarification. If NULL is passed as *descriptions*, no descriptions are returned.

Return Value:

RaDeKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Remarks:

This function will automatically allocate the storage required for the serial number and description lists and the associated character strings. If valid pointers are passed in, the existing storage (as allocated by a previous call to *RaDeKL_ListRadars*) will first be de-allocated before the new lists are allocated. This way the designer does not need to bother maintaining these lists (i.e. simplifying re-scanning of connected radar units), as long as the pointers are *initially* set to NULL. See examples below for clarification. Any remaining lists will be de-allocated when the program terminates. If the design requires an explicit de-allocation of the lists, use *RaDeKL_ListRadarsCleanup* described below.

Example 1:

List all available radar units and display the information obtained.

```
ULONG       status, i;
DWORD       numdevs;
static char **snum = NULL, **desc = NULL;   // or make these global

status = RaDeKL_ListRadars (&numdevs, &snum, &desc);
if (status != RaDeKL_OK)
{
    printf ("Unable to list devices: %s\n", RaDeKL_GetStatusText (status));
    return;
}

printf ("Number of devices detected: %d\n", numdevs);
for (i = 0; i < numdevs; i++)
    printf ("Device %d, Serial: %s, Description: %s\n", i, snum[i], desc[i]);
```

Example 2:

Alternately, if we don't care about the number of devices or their descriptions, the following code segment will return only the (null-terminated) list of serial numbers:

```
ULONG       status;
static char **snum = NULL;  // or make this global
char        **s;

status = RaDeKL_ListRadars (NULL, &snum, NULL);
if (status != RaDeKL_OK)
{
    printf ("Unable to list devices: %s\n", RaDeKL_GetStatusText (status));
    return;
}

for (s = snum; *s; s++)
    printf ("Serial Number: %s\n", *s);
```

RaDeKL_ListRadarsCleanup

De-allocates the memory that was allocated in a previous call to *RaDeKL_ListRadars*. Normally, this is not required as each successive call to *RaDeKL_ListRadars* de-allocates these lists and Windows will do the final de-allocation when the program terminates. However, if the design calls for an explicit de-allocation, this function can be used.

Format:

void **RaDeKL_ListRadarsCleanup** (char ***list*);

Parameters:

list	Pointer to a (null-terminated) list of character strings to de-allocate.

Return Value:

No return value. This function cannot fail.

Example:

List all available radar units and display the information obtained.

```
ULONG       status, i;
DWORD       numdevs;
static char **snum = NULL, **desc = NULL;   // or make these global

status = RaDeKL_ListRadars (&numdevs, &snum, &desc);
if (status != RaDeKL_OK)
{
    printf ("Unable to list devices: %s\n", RaDeKL_GetStatusText (status));
    return;
}

printf ("Number of devices detected: %d\n", numdevs);
for (i = 0; i < numdevs; i++)
    printf ("Device %d, Serial: %s, Description: %s\n", i, snum[i], desc[i]);

// De-allocate the lists
RaDeKL_ListRadarsCleanup (snum); snum = NULL;
RaDeKL_ListRadarsCleanup (desc); snum = NULL;
```

RaDeKL_OpenRadar

Open a RaDeKL Radar device as identified by its serial number. Serial numbers are unique for each specific unit. A list of all available RaDeKL Radar devices and their serial numbers is obtained by calling *RaDeKL_ListRadars*.

Format:

ULONG **RaDeKL_OpenRadar** (RaDeKL_HANDLE **handle_ptr*, char **serial_number*);

Parameters:

handle_ptr	Pointer to a RaDeKL_HANDLE to receive the handle for the opened device. All further operations on that device will use this handle.
serial_number	Pointer to a (null-terminated) character string containing the serial number of the device to open. A list of all available RaDeKL Radar devices and their serial numbers is obtained by calling *RaDeKL_ListRadars*.

Return Value:

RaDeKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Open (and close) the first radar unit found.

```
ULONG                   status;
DWORD                   numdevs;
static char             **snum = NULL, **desc = NULL;   // or make these global
static RaDeKL_HANDLE    handle = NULL;                  // or make this global

status = RaDeKL_ListRadars (&numdevs, &snum, &desc);
if (status != RaDeKL_OK)
{
    printf ("Unable to list devices: %s\n", RaDeKL_GetStatusText (status));
    return;
}

if (numdevs == 0)
{
    printf ("No devices available\n");
    return;
}

status = RaDeKL_OpenRadar (&handle, snum[0]);
if (status != RADEKL_OK)
{
    printf ("Open failed: %s\n", RaDeKL_GetStatusText (status));
    return;
}
// Do some work and then close the radar and set the handle to NULL
RaDeKL_CloseRadar (handle);
handle = NULL;
```

RaDeKL_CloseRadar

Close a RaDeKL Radar device previously opened by a call to *RaDeKL_OpenRadar*.

Format:

ULONG **RaDeKL_CloseRadar** (RaDeKL_HANDLE *handle*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Remarks:

It is highly recommended that you set the *handle* to *NULL* after calling this function to avoid accidental use of a closed handle. See the example for *RaDeKL_OpenRadar* above.

Example:

See the example for *RaDeKL_OpenRadar* above.

RaDeKL_GetDeviceInfo

Get the device-specific information of a RaDeKL Radar associated with the handle.

Format:

ULONG **RaDeKL_GetDeviceInfo** (RaDeKL_HANDLE *handle*, RaDeKL_DEVICEINFO **info*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
info	Pointer to a RaDeKL_DEVICEINFO structure. See the appendix for field definitions of this structure.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Remarks:

RaDeKL_GetDeviceInfo should be called after *RaDeKL_OpenRadar* to obtain device-version-specific information. The currently available RaDeKL units all have a *resolution* of 1 foot (= 2), 256 range bins (with a data range of 0 to 32), 32 DAC threshold registers (with a data range of 20 to 227). This may change in future units and the info returned by *RaDeKL_GetDeviceInfo* should be used in your application instead of hard-coding these values.

Example:

List device-specific information.

```
ULONG               status;
RaDeKL_HANDLE       handle;
RaDeKL_DEVICEINFO   info;

// Assume we have an open radar with a valid handle

status = RaDeKL_GetDeviceInfo (handle, &info);
// Check status . . .

// Print information
printf ("Serial number:  %s\n",  info.ft_serial_number);// String
printf ("Description:    %s\n",  info.ft_description);  // String
printf ("Vendor ID:      %4X\n", info.ft_vendor_id);    // 16-bit hex
printf ("Product ID:     %4X\n", info.ft_product_id);   // 16-but hex
printf ("Resolution:     %d\n",  info.resolution);      // 1 = 6-inch, 2 = 1-foot
printf ("Range bins:     %d\n",  info.range_bins);      // Currently 256
printf ("Range bin max:  %d\n",  info.range_bin_max);   // Currently 32
printf ("Thresholds:     %d\n",  info.thresholds);      // Currently 32
printf ("Threshold min:  %d\n",  info.threshold_min);   // Currently 20
printf ("Threshold max:  %d\n",  info.threshold_max);   // Currently 227
printf ("Version:        v%d.%d\n",  info.version_id_major, info.version_id_minor);
```

RaDeKL_ResetRadar

Perform a reset on the radar, resetting the device to factory settings and purging the send and receive buffers.

> ***NOTE:*** Resetting the radar causes all radar registers to be reset to factory default values. That means that the settings for DAC thresholds, TX and RX attenuation, range delay and detection intervals may no longer be the same as what the application thinks they are. It is strongly recommended to re-send all pertinent register values to the radar after calling this function to ensure that the application and the actual radar unit are in sync.

Format:

ULONG **RaDeKL_ResetRadar** (RaDeKL_HANDLE *handle*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Perform a reset on a RaDeKL Radar device.

```
ULONG           status;
RaDeKL_HANDLE   handle;

// Assume we have an open radar with a valid handle

status = RaDeKL_ResetRadar (handle);
// Check status . . .
```

RaDeKL_FlushIO

Flushes (purges) the transmit and receive buffers on the USB port the radar is connected to. Note that this does not necessarily flush the data on the FPGA radar processor chip ports. This function is intended only to flush the data waiting on the USB chip ports. To properly reset the RaDeKL radar device, use *RaDeKL_ResetRadar* instead.

Format:

ULONG **RaDeKL_FlushIO** (RaDeKL_HANDLE *handle*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Flush the USB transmit and receive ports.

```
ULONG           status;
RaDeKL_HANDLE   handle;

// Assume we have an open radar with a valid handle

status = RaDeKL_FlushIO (handle);
// Check status . . .
```

Part IV: Radar Parameter Functions

RaDeKL_SetThresholds

Set the DAC Threshold values in the radar device.

> ***NOTE:*** There are currently 32 DAC Threshold registers with permissible values in the range from 20 to 227, but the actual number of registers and permissible values should be obtained by calling *RaDeKL_GetDeviceInfo* and using *thresholds*, *threshold_min* and *threshold_max*, as these might change with future versions of the radar unit.

The default values for the 32 DAC Threshold registers start at 20 (DAC 1) and are evenly spread up to 227 (DAC 32).

Format:

ULONG **RaDeKL_SetThresholds** (RaDeKL_HANDLE *handle*, BYTE **thresholds*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
thresholds	A BYTE array of sufficient size to hold all DAC Threshold register values (currently 32).

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Add 10 to all DAC Threshold values (assuring that the max is not exceeded).

```
ULONG               status;
RaDeKL_HANDLE       handle;
RaDeKL_DEVICEINFO   info;
BYTE                thresholds[256];    // Declare a reasonably large number

// Assume we have an open radar with a valid handle
status = RaDeKL_GetDeviceInfo (handle, &info);
// Check status . . .

// Get the current DAC Threshold values
status = RaDeKL_GetThresholds (handle, thresholds);
// Check status . . .

// Add 10 to ALL registers (limited by threshold_max) and print the values
for (i = 0; i < info.thresholds; i++)
{
    thresholds[i] = min (thresholds[i]+10, info.threshold_max);
    printf ("DAC Threshold register %2d changed to %d\n", i, thresholds[i]);
}

// Set the modified DAC Threshold values
status = RaDeKL_SetThresholds (handle, thresholds);
// Check status . . .
```

RaDeKL_GetThresholds

Get the DAC Threshold values from the radar device.

> ***NOTE:*** There are currently 32 DAC Threshold registers with permissible values in the range from 20 to 227, but the actual number of registers and permissible values should be obtained by calling *RaDeKL_GetDeviceInfo* and using *thresholds*, *threshold_min* and *threshold_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_GetThresholds** (RaDeKL_HANDLE *handle*, BYTE **thresholds*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
thresholds	A BYTE array of sufficient size to hold all DAC Threshold register values (currently 32).

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

See the example given for *RaDeKL_SetThresholds* above.

RaDeKL_SetTransmitAttenuation

Set the TRANSMIT POWER ATTENUATION register in the radar device.

> ***NOTE:*** Transmit attenuation settings can currently be in the range from 0 to 63, but the actual range of permissible settings should be obtained by calling *RaDeKL_GetDeviceInfo* and using *tx_atten_min* and *tx_atten_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_SetTransmitAttenuation** (RaDeKL_HANDLE *handle*, BYTE *attenuation*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

attenuation A BYTE indicating the attenuation to be set. Permissible *attenuation* values are in the range from 0 to 63. Reference values are:

Value	Attenuation
63	0 dB (default)
57	-3 dB
51	-6 dB
43	-10 dB

The attenuation (in dB) or the register value are calculated as:

attenuation_dB = (register_value – 63) / 2

and

register_value = (2 * attenuation_dB) + 63

Note: attenuations are usually **negative** values! *Attenuation_dB* in the above calculations must therefore be a negative number.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Set the TRANSMIT ATTENUATION register to a value of 33 (-10 dB).

```
ULONG           status;
RaDeKL_HANDLE   handle;

// Assume we have an open radar with a valid handle

status = RaDeKL_SetTransmitAttenuation (handle, 33);
// Check status . . .
```

RaDeKL_GetTransmitAttenuation

Get the TRANSMIT POWER ATTENUATION register value from the radar device.

> ***NOTE:*** Transmit attenuation settings can currently be in the range from 0 to 63, but the actual range of permissible settings should be obtained by calling *RaDeKL_GetDeviceInfo* and using *tx_atten_min* and *tx_atten_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_GetTransmitAttenuation** (RaDeKL_HANDLE *handle*, BYTE **attenuation*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

attenuation Pointer to a BYTE to receive the attenuation value. Possible *attenuation* values are in the range from 0 to 63. Reference values are:

Value	Attenuation
63	0 dB (default)
57	-3 dB
51	-6 dB
43	-10 dB

The attenuation (in dB) or the register value are calculated as:

attenuation_dB = (register_value – 63) / 2

and

register_value = (2 * attenuation_dB) + 63

Note: attenuations are usually **negative** values! *Attenuation_dB* in the above calculations must therefore be a negative number.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Read the TRANSMIT ATTENUATION register and display its value.

```
ULONG           status;
RaDeKL_HANDLE   handle;
BYTE            attenuation;

// Assume we have an open radar with a valid handle

status = RaDeKL_GetTransmitAttenuation (handle, &attenuation);
// Check status . . .

printf ("The TRANSMIT ATTENUATION is currently set to %d\n", attenuation);
```

RaDeKL_SetReceiveAttenuation

Set the RECEIVE ATTENUATION register in the radar device.

> ***NOTE:*** Receive attenuation settings can currently be in the range from 0 to 255, but the actual range of permissible settings should be obtained by calling *RaDeKL_GetDeviceInfo* and using *rx_atten_min* and *rx_atten_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_SetReceiveAttenuation** (RaDeKL_HANDLE *handle*, BYTE *attenuation*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

attenuation A BYTE indicating the attenuation to be set. Permissible *attenuation* values are in the range from 0 to 255. Reference values are:

Value	Attenuation
0	0 dB (default)
31	-5 dB
81	-10 dB
125	-15 dB
155	-20 dB
180	-30 dB

For the receive attenuation there is no function to convert between register value and attenuation. The above values are derived from experimentation. The user may use any register value between 0 and 255 and use the above table as a rough guide for the resulting attenuation.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Set the RECEIVE ATTENUATION register to a value of 91 (-10 dB).

```
ULONG           status;
RaDeKL_HANDLE   handle;

// Assume we have an open radar with a valid handle

status = RaDeKL_SetReceiveAttenuation (handle, 91);
// Check status . . .
```

RaDeKL_GetReceiveAttenuation

Get the RECEIVE ATTENUATION register value from the radar device.

> ***NOTE:*** Receive attenuation settings can currently be in the range from 0 to 255, but the actual range of permissible settings should be obtained by calling *RaDeKL_GetDeviceInfo* and using *rx_atten_min* and *rx_atten_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_GetReceiveAttenuation** (RaDeKL_HANDLE *handle*, BYTE **attenuation*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

attenuation Pointer to a BYTE to receive the attenuation value. Possible *attenuation* values are in the range from 0 to 255. Reference values are:

Value	Attenuation
0	0 dB (default)
31	-5 dB
81	-10 dB
125	-15 dB
155	-20 dB
180	-30 dB

For the receive attenuation there is no function to convert between register value and attenuation. The above values are derived from experimentation. The user may use any register value between 0 and 255 and use the above table as a rough guide for the resulting attenuation.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Read the RECEIVE ATTENUATION register and display its value.

```
ULONG           status;
RaDeKL_HANDLE   handle;
BYTE            attenuation;

// Assume we have an open radar with a valid handle

status = RaDeKL_GetReceiveAttenuation (handle, &attenuation);
// Check status . . .

printf ("The RECEIVE ATTENUATION is currently set to %d\n", attenuation);
```

RaDeKL_SetRangeDelay

Set the RANGE and DELAY registers in the radar device. These two registers shift the return data by a certain number of range bins. To simplify implementations, this function accepts a RangeDelay in feet and converts it to the required register values. The default value is 0 feet.

Format:

ULONG **RaDeKL_SetRangeDelay** (RaDeKL_HANDLE *handle*,
DWORD *delay_feet*, DWORD **actual_delay_feet*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
delay_feet	A DWORD indicating the requested delay value in feet. Do not use a value larger than the maximum range of the radar device.
actual_delay_feet	Pointer to a DWORD to receive the actual delay set. This may be NULL, in which case no actual delay is returned. The actual delay may differ from the requested one, since the hardware allows delays only in certain increments (8 feet in the current model). If necessary, the requested value is rounded down to the nearest increment.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Set the RangeDelay to 32 feet.

```
ULONG           status;
RaDeKL_HANDLE   handle;
DWORD           actual;

// Assume we have an open radar with a valid handle

status = RaDeKL_SetRangeDelay (handle, 32, &actual);
// Check status . . .

printf ("Range Delay set to %d feet\n", actual);
```

RaDeKL_GetRangeDelay

Get the RangeDelay from the radar device by reading the RANGE and DELAY registers and converting the values to feet.

Format:

ULONG **RaDeKL_GetRangeDelay** (RaDeKL_HANDLE *handle*, DWORD **delay_feet*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
delay_feet	Pointer to a DWORD to receive the delay value in feet.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Read the RangeDelay and display its value.

```
ULONG           status;
RaDeKL_HANDLE   handle;
DWORD           delay;

// Assume we have an open radar with a valid handle

status = RaDeKL_GetRangeDelay (handle, &delay);
// Check status . . .

printf ("The DELAY is currently set to %d feet\n", delay);
```

RaDeKL_SetInterval

Set the automatic detection interval for *continuous* detection mode. When the radar is in continuous mode, a new detection is performed each time this interval expires. The default setting is INTERVAL_50_MS.

> ***NOTE:*** There are currently 5 distinct interval settings (see table below), but the actual range of permissible settings should be obtained by calling *RaDeKL_GetDeviceInfo* and using *interval_min* and *interval_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_SetInterval** (RaDeKL_HANDLE *handle*, BYTE *interval*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

interval A BYTE indicating the requested interval. Permissible *interval* values are:

Symbolic Constant	Duration	Value
INTERVAL_1_SEC	1 second	0
INTERVAL_500_MSEC	500 milliseconds	1
INTERVAL_250_MSEC	250 milliseconds	2
INTERVAL_100_MSEC	100 milliseconds	3
INTERVAL_50_MSEC (default)	50 milliseconds	4

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Set the continuous detection interval to 100 milliseconds.

```
ULONG           status;
RaDeKL_HANDLE   handle;

// Assume we have an open radar with a valid handle

status = RaDeKL_SetInterval (handle, INTERVAL_100_MS);
// Check status . . .
```

RaDeKL_GetInterval

Get the automatic detection interval for *continuous* detection mode in milliseconds.

> ***NOTE:*** There are currently 5 distinct interval settings (see table below), but the actual range of permissible settings should be obtained by calling *RaDeKL_GetDeviceInfo* and using *interval_min* and *interval_max*, as these might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_GetInterval** (RaDeKL_HANDLE *handle*, BYTE **interval*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

interval Pointer to a BYTE to receive the interval value. Possible *interval* values are:

Symbolic Constant	Duration	Value
INTERVAL_1_SEC	1 second	0
INTERVAL_500_MSEC	500 milliseconds	1
INTERVAL_250_MSEC	250 milliseconds	2
INTERVAL_100_MSEC	100 milliseconds	3
INTERVAL_50_MSEC (default)	50 milliseconds	4

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Read the continuous detection interval and display its value.

```
ULONG           status;
RaDeKL_HANDLE   handle;
BYTE            interval;

// Assume we have an open radar with a valid handle

status = RaDeKL_GetInterval (handle, &interval);
// Check status . . .

printf ("The interval for continuous detection is currently set to ");
switch (interval)
{
    case INTERVAL_1_SEC:  printf ("1 second\n"); break;
    case INTERVAL_500_MS: printf ("500 ms\n");   break;
    case INTERVAL_250_MS: printf ("250 ms\n");   break;
    case INTERVAL_100_MS: printf ("100 ms\n");   break;
    case INTERVAL_50_MS:  printf ("50 ms\n");    break;
}
```

RaDeKL_SimulatorMode

The RaDeKL Radar API provides a transparent radar simulation mode. This allows programs using this API to be tested *without* the need to have an actual RaDeKL Radar unit present. By default, Simulator Mode is *disabled*, causing communications to occur with an actual radar device. In situations where an actual radar unit is unavailable for testing of the software, simulation mode may be enabled, resulting in the proper and expected behavior of all API functions with these considerations:

1. *RaDeKL_ListRadars* will detect four simulated radar units with serial numbers "SIM001", "SIM002", "SIM003" and "SIM004" and a description of "RaDeKL Simulator".
2. Any RaDeKL API function may be called for these simulated radar units, including detection functions.
3. Detection will result in range bin data representing a dampened sine wave, rotating over time (or successive detections) for SIM001 and other diagnostic waveforms for SIM002 - SIM004.
4. Since there is no physical communication to an actual radar unit, some error paths in the program may not be sufficiently tested using Simulator Mode.

Format:

bool **RaDeKL_SimulatorMode** (bool *enable*);

Parameters:

enable Flag to enable (*true*) or disable (*false*) the RaDeKL Simulator Mode.

Return Value:

The *previous* value of the RaDeKL Simulator Mode.

Remarks:

Normally, this function would be called at the start of your program, before any potential radar units have been opened. However, if a program switches dynamically between Simulator and actual radar modes, ensure that any open radar devices (real or simulated) are closed first using *RaDeKL_CloseRadar*. Toggling Simulator Mode will cause *RaDeKL_ListRadars* to return a different list of available radar devices. Therefore a call to *RaDeKL_SimulatorMode* should always be followed by a call to *RaDeKL_ListRadars* to refresh the list of available devices.

Example:

Enable the RaDeKL Simulator mode.

```
ULONG       status;
DWORD       numdevs;
static char **snum = NULL, **desc = NULL;   // or make these global

RaDeKL_SimulatorMode (true);
printf ("The RaDeKL Simulation Mode is now enabled\n");

status = RaDeKL_ListRadars (&numdevs, &snum, &desc);
// Check status . . .
```

Part V: Radar Detection Functions

RaDeKL_DetectSingle

Perform a *single* radar detection and return the range bin data.

> ***NOTE:*** The number of range bins is currently 256 with possible data values between 0 and 32, but the actual number of range bins and the range of their possible values should be obtained by calling *RaDeKL_GetDeviceInfo* and using *range_bins*, *range_bin_min* and *range_bin_max*, as they might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_DetectSingle** (RaDeKL_HANDLE *handle*, BYTE **data*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
data	Pointer to a BYTE array to receive the radar return data.
	Note: This array is ***not*** null-terminated, since zeroes may appear in the data.
	Note: the size of this array and the maximum value for each range bin are currently fixed at 256 and 32, respectively, but might change with future versions. See notes above.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Initiate a *single* detection and print the range bin data.

```
ULONG               status, i;
RaDeKL_HANDLE       handle;
RaDeKL_DEVICEINFO   info;
BYTE                data[2048];     // Make this sufficiently large

// Assume we have an open radar with a valid handle

status = RaDeKL_GetDeviceInfo (handle, &info);
// Check status . . .

status = RaDeKL_DetectSingle (handle, data);
// Check status . . .

for (i = 0; i < info.range_bins; i++)
    printf ("Range bin [%3d] contains: %d\n", i, data[i]);
```

RaDeKL_StartContinuousDetection

Start continuous detection, with a new detection occurring each interval set with *RaDeKL_SetInterval*. This function is intended to be used in conjunction with *RaDeKL_ReadDetectionData* and *RaDeKL_StopContinuousDetection*.

> ***Note:*** Unless there is an initially known fixed number of detections to be performed, continuous detection may require a separate thread to execute *RaDeKL_ReadDetectionData*, or otherwise it may be impossible to issue *RaDeKL_StopContinuousDetection* since *RaDeKL_ReadDetectionData* will be in a tight loop waiting for the next detection data to arrive. See examples below for clarification.
>
> ***Note:*** While in *continuous* detection mode, do not issue any other register read/write functions (except RaDeKL_StopContinuousDetection) as their processing will interfere with the continuous data stream returned by the RaDeKL Radar device. See *Usage Warnings* towards the top of this document for clarification.

Format:

ULONG **RaDeKL_StartContinuousDetection** (RaDeKL_HANDLE *handle*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example 1:

Process 100 continuous radar detections.

```
ULONG           status, count;
RaDeKL_HANDLE   handle;
BYTE            data[2048]; // Make this sufficiently large

// Assume we have an open radar with a valid handle
status = RaDeKL_StartContinuousDetection (handle);
// Check status . . .

count = 100;
while (count--)
{
    status = RaDeKL_ReadDetectionData (handle, data);
    // Check status . . .

    // Do something with the data . . .
    //
}
status = RaDeKL_StopContinuousDetection (handle);
// Check status . . .

// Flush the buffers
status = RaDeKL_FlushIO (handle);
// Check status . . .
```

Example 2:

Process continuous radar detections until user stops. This requires 2 threads, one to handle the GUI and the other to read and process the radar range bin data.

```
// Global data
RaDeKL_HANDLE   handle;
HANDLE          g_hthread;
BOOL            running = false;
```

>>> Thread 1 (GUI):

```
// Assume we have an open radar with a valid handle AND a secondary thread

// User signals to START continuous detection
status = RaDeKL_StartContinuousDetection (handle);
// Check status . . .
// Wake up secondary thread
running = true;
ResumeThread (g_hthread);

// Return to the GUI message loop . . .

// User signals to STOP continuous detection
running = false;
status = RaDeKL_StopContinuousDetection (handle);
// Check status . . .

// Flush the buffers
status = RaDeKL_FlushIO (handle);
// Check status . . .
```

>>> Thread 2 (read and process loop):

```
ULONG           status;
BYTE            data[2048]; // Make this sufficiently large

// Assume we have an open radar with a valid handle

while (true)    // Loop this thread forever until the program terminates
{
    while (running) // Loop until the GUI sets running to false
    {
        status = RaDeKL_ReadDetectionData (handle, data);

        // Check status. Ignore a timeout if running is false
        if ((status == RADEKL_OK) || ((!running) && (status == RADEKL_READ_TIMEOUT)))
        {
            // Do something with the data . . .
        }
        else
        {
            printf ("Bad read: %s\n", RaDeKL_GetStatusText (status));
            running = false;
            break;
        }
    }
    // No longer running - go to sleep
    SuspendThread (GetCurrentThread ());
}
```

RaDeKL_StopContinuousDetection

Stop continuous detection. This function is intended to be used in conjunction with *RaDeKL_StartContinuousDetection* and *RaDeKL_ReadDetectionData*.

Format:

ULONG **RaDeKL_StopContinuousDetection** (RaDeKL_HANDLE *handle*);

Parameters:

handle RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

See the examples for *RaDeKL_StartContinuousDetection* above.

RaDeKL_ReadDetectionData

Read and return the range bin data of a single detection in *continuous* mode. This function is intended to be used in conjunction with *RaDeKL_StartContinuousDetection* and *RaDeKL_StopContinuousDetection*.

> ***NOTE:*** The number of range bins is currently 256 with possible data values between 0 and 32, but the actual number of range bins and the range of their possible values should be obtained by calling *RaDeKL_GetDeviceInfo* and using *range_bins*, *range_bin_min* and *range_bin_max*, as they might change with future versions of the radar unit.

Format:

ULONG **RaDeKL_ReadDetectionData** (RaDeKL_HANDLE *handle*, BYTE **data*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
data	Pointer to a BYTE array to receive the radar return data. ***Note:*** This array is ***not*** null-terminated, since zeroes may appear in the data. ***Note:*** the size of this array and the maximum value for each range bin are currently fixed at 256 and 32, respectively, but might change with future versions. See notes for *RaDeKL_DetectSingle*.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

See the examples for *RaDeKL_StartContinuousDetection* above.

Part VI: Radar Register Functions

RaDeKL_WriteCheckRegister

> ***Note:*** This is a low-level function and should only be used if you are familiar with the RaDeKL Radar registers and their layouts. Normal operation of the RaDeKL Radar should require only the use of the high-level functions described above.

Write a data BYTE into one of RaDeKL Radar's registers. The data is then read back and compared to ensure that it was, in fact, written correctly.

> ***Note:*** This function *reads* the register back to ensure that it was written correctly. *Do not* use this function on command registers that have write-only bits (such as REGISTER_DETECT). Use *RaDeKL_WriteRegister* instead.

Format:

ULONG **RaDeKL_WriteCheckRegister** (RaDeKL_HANDLE *handle*, WORD *addr*, BYTE *value*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
addr	16-bit address of the register to modify. See the appendix for register definitions.
value	8-bit value to put into the requested register.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Write a value of 50 into register at address 8 (REGISTER_DAC1).

```
ULONG           status;
RaDeKL_HANDLE   handle;

// Assume we have an open radar with a valid handle

status = RaDeKL_WriteCheckRegister (handle, REGISTER_DAC1, 50);
// Check status . . .
```

RaDeKL_WriteRegister

> ***Note:*** This is a low-level function and should only be used if you are familiar with the RaDeKL Radar registers and their layouts. Normal operation of the RaDeKL Radar should require only the use of the high-level functions described above.

Write a data BYTE into one of RaDeKL Radar's registers. No checking is performed to see if the data was written correctly, making this function suitable for write-only registers. If verification is required, use *RaDeKL_WriteCheckRegister* instead.

Format:

ULONG **RaDeKL_WriteRegister** (RaDeKL_HANDLE *handle*, WORD *addr*, BYTE *value*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
addr	16-bit address of the register to modify. See the appendix for register definitions.
value	8-bit value to put into the requested register.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Write DETECT_SINGLE (1) into REGISTER_DETECT at address 1. This triggers a single detection and the transmission of the range bin data.

```
ULONG           status;
RaDeKL_HANDLE   handle;
BYTE            data[2048]; // Make this sufficiently large

// Assume we have an open radar with a valid handle

status = RaDeKL_WriteRegister (handle, REGISTER_DETECT, DETECT_SINGLE);
// Check status . . .

// Read the range bin data
status =  RaDeKL_ReadDetectionData (handle, data);
// Check status . . .
```

RaDeKL_ReadRegister

> ***Note:*** This is a low-level function and should only be used if you are familiar with the RaDeKL Radar registers and their layouts. Normal operation of the RaDeKL Radar should require only the use of the high-level functions described above.

Read a sequence of one or more RaDeKL Radar registers.

Format:

ULONG **RaDeKL_ReadRegister** (RaDeKL_HANDLE *handle*, WORD *addr*,
BYTE *count*, BYTE **values*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
addr	16-bit address of the register to modify. See the appendix for register definitions.
count	8-bit count of successive registers to read.
values	Pointer to an BYTE-array to receive the values of the requested register(s).

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Read and print the value stored in register REGISTER_ID_VERS (0)

```
ULONG           status;
RaDeKL_HANDLE   handle;
BYTE            version;

// Assume we have an open radar with a valid handle

status = RaDeKL_ReadRegister (handle, REGISTER_ID_VERS, 1, &version);
// Check status . . .

printf ("The version register contains %02X (major = %d, minor = %d)\n",
        version, version >> 4, version & 0xF);
```

Part VII: Low-Level Data I/O Functions

RaDeKL_SendCommand

Note: This is a low-level function and should only be used if you are familiar with the RaDeKL Radar I/O features. Normal operation of the RaDeKL Radar should require only the use of the high-level functions described above.

Send a command to the RaDeKL Radar. The command must be a *read* register or a *write register* command in the following form:

Read Register (always 5 bytes): <0x72><*addr-hi*><*addr-lo*><*qty*><0xFF>

Write Register (always 5 bytes): <0x77><*addr-hi*><*addr-lo*><*val*><0xFF>

Where: <*addr-hi*> is the high byte of the 16-bit register address.
<*addr-lo*> is the low byte of the 16-bit register address.
(See the appendix for register definitions)

<*qty*> is an 8-bit count of registers to read (*read* command only).

<*val*> is an 8-bit value to write to the register (*write* command only).

Format:

ULONG **RaDeKL_SendCommand** (RaDeKL_HANDLE *handle*, BYTE **cmd*,
DWORD *bytes_to_send*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
cmd	Pointer to the 5-byte command.
bytes_to_send	Number of bytes to write (should always be 5, unless a future version extends the I/O interface).

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example:

Write a value of 50 into register at address 8 (REGISTER_DAC1).

```
ULONG           status;
RaDeKL_HANDLE   handle;
BYTE            cmd[5] = {CMD_WRITE, 0, REGISTER_DAC1, 50, CMD_TERM};

// Assume we have an open radar with a valid handle

status = RaDeKL_SendCommand (handle, cmd, 5);
// Check status . . .
```

RaDeKL_ReceiveResponse

Note: This is a low-level function and should only be used if you are familiar with the RaDeKL Radar I/O features. Normal operation of the RaDeKL Radar should require only the use of the high-level functions described above.

Receive a response from the radar by reading a specified number of bytes. There are two types of responses, *returned register values* and a *set of range bins* resulting from a detection request. They are in the form:

Returned register values (<number of registers requested> + 3 bytes):
<*addr-hi*><*addr-lo*><*data-1*>...<*data-n*><0xFF>

Range bins from detection (<number of range bins> + 4 bytes [currently 260 bytes]):
<0xEA><0xEA><0xEA><*bin-1*>…<*bin-256*><0xFF>

Where: <*addr-hi*> is the high byte of the 16-bit register address.
<*addr-lo*> is the low byte of the 16-bit register address.
(See the appendix for register definitions)

<*data-n*> is a series of 8-bit register values (*returned register values* only).

<*bin-n*> is an 8-bit value from the [n]th range bin (*detection* only).

RaDeKL_ReceiveResponse returns the entire response, including the possible prefix (0xEA) and the terminator (0xFF).

Format:

ULONG **RaDeKL_ReceiveResponse** (RaDeKL_HANDLE *handle*, BYTE **response*,
DWORD *bytes_requested*, DWORD **bytes_received*);

Parameters:

handle	RaDeKL_HANDLE as returned by a call to RaDeKL_OpenRadar.
response	Pointer to a BYTE array to receive the data returned.
bytes_requested	DWORD containing the number of bytes to read.
bytes_received	Pointer to a DWORD to receive the number of bytes read.

Return Value:

RADEKL_OK (0) if successful, a non-zero status otherwise. See *RaDeKL_GetStatusText* for codes.

Example 1:

Read the value of the register at address 7 (REGISTER_DAC1).

```
ULONG           status;
DWORD           bytes_received;
RaDeKL_HANDLE   handle;
BYTE            cmd[5] = {CMD_READ, 0, REGISTER_DAC1, 1, CMD_TERM};
BYTE            response [1024];    // Make this sufficiently large

// Assume we have an open radar with a valid handle

status = RaDeKL_SendCommand (handle, cmd, 5);
// Check status . . .

status = RaDeKL_ReceiveResponse (handle, response, 4, &bytes_received);
// Check status . . . and ensure that bytes_received == 4

printf ("The value at register DAC 1 is %d\n", response[2]);
```

Example 2:

Issue a single detection request and read the range bins returned.

```
ULONG           status;
DWORD           bytes_received;
RaDeKL_HANDLE   handle;
BYTE            cmd[5] = {CMD_WRITE, 0, REGISTER_DETECT, DETECT_SINGLE, CMD_TERM};
BYTE            response [1024];    // Make this sufficiently large

// Assume we have an open radar with a valid handle

status = RaDeKL_SendCommand (handle, cmd, 5);
// Check status . . .

status = RaDeKL_ReceiveResponse (handle, response, 260, &bytes_received);
// Check status . . . and ensure that cnt == 260 (number of range bins + 4)

// The range bin data is now in response[3] thru response [258].
// response [0], [1] and [2] contain 0xEA and response[259] 0xFF (terminator).
```

Part VIII: Appendix

RaDeKL_DEVICEINFO Field Definitions

The following list provides a list of the fields available in the *RaDeKL_DEVICEINFO* structure, obtained by calling *RaDeKL_GetDeviceInfo*. For the actual source-code of the structure definition, see the *RaDeKLAPI.H Header File Listing* at the end of the Appendix.

Register Name	Type	Description
FTDI USB Chip specific		Please use these (except *Serial Number* and *Device Description*) only if you are familiar with the FTDI USB Chip interface.
ft_handle	void *	Handle to the underlying USB device (do not use unless you are familiar with the FTDI USB chip API FTD2XX.h)
ft_device_type	ULONG	USB device type (see FTD2XX.h for details)
ft_serial_number	char[16]	Serial number (null-terminated)
ft_description	char[64]	Device description (null-terminated)
ft_device_id	DWORD	USB Chip device id (see FTD2XX.h for details)
ft_product_id	WORD	USB Chip product id (see FTD2XX.h for details)
ft_vendor_id	WORD	USB Chip vendor id (see FTD2XX.h for details)
RaDeKL Radar specific		**Please use** these values in your code instead of hard-coding them as configurations and ranges in future versions of the radar may change.
resolution	BYTE	Resolution (currently RESOLUTION_1FOOT = 2) Use this value to determine the actual distance represented between successive range bin data samples. (See *RaDeKLAPI.H* for details)
range_bins	WORD	The number of range bins available (currently 256)
range_bin_min	BYTE	Minimum value of a range bin (currently 0)
range_bin_max	BYTE	Maximum value of a range bin (currently 32)
thresholds	BYTE	The number of Threshold registers (currently 32)
threshold_min	BYTE	Minimum value of a Threshold register (currently 20)
threshold_max	BYTE	Maximum value of a Threshold register (currently 227)
tx_atten_min	BYTE	Minimum value for the Transmit Attenuation register (currently 0)
tx_atten_max	BYTE	Maximum value for the Transmit Attenuation register (currently 63)
rx_atten_min	BYTE	Minimum value for the Receive Attenuation register (currently 0)
rx_atten_max	BYTE	Maximum value for the Receive Attenuation register (currently 255)
interval_min	BYTE	Minimum value for the Interval register (currently 0)
interval_max	BYTE	Maximum value for the Interval register (currently 4)
Interval	BYTE	Current interval setting. Do not use this field! It is used for internal purposes and cannot be relied upon within the context of this API.
version_id	BYTE	Full 8-bit version number of the RaDeKL Radar device
version_id_minor	4 bits	4-bit Minor version number of the RaDeKL Radar device. This field overlays the low-order 4 bits of *version_id.*
version_id_major	4 bits	4-bit Major version number of the RaDeKL Radar device. This field overlays the high-order 4 bits of *version_id.*

RaDeKL Radar Register Definitions

The following list provides the *current* definition of the RaDeKL Radar registers. These definitions may change with future versions of the radar device.

Register Name	Address	Description
REGISTER_ID_VERS	0	Version ID register
REGISTER_DETECT	1	Detection command register
REGISTER_TID_TIME	2	Interval for continuous detections
REGISTER_RF_CONTROL	3	RF Control register
REGISTER_TX_ATTEN	4	Transmitter attenuation register
REGISTER_RX_ATTEN	5	Receiver attenuation register
REGISTER_RANGE	6	RANGE register (shift by 512 range bins)
REGISTER_DELAY	7	DELAY register (shift by 8 range bins)
REGISTER_DAC1	8	DAC Threshold setting 1
REGISTER_DAC2	9	DAC Threshold setting 2
REGISTER_DAC3	10	DAC Threshold setting 3
REGISTER_DAC4	11	DAC Threshold setting 4
REGISTER_DAC5	12	DAC Threshold setting 5
REGISTER_DAC6	13	DAC Threshold setting 6
REGISTER_DAC7	14	DAC Threshold setting 7
REGISTER_DAC8	15	DAC Threshold setting 8
REGISTER_DAC9	16	DAC Threshold setting 9
REGISTER_DAC10	17	DAC Threshold setting 10
REGISTER_DAC11	18	DAC Threshold setting 11
REGISTER_DAC12	19	DAC Threshold setting 12
REGISTER_DAC13	20	DAC Threshold setting 13
REGISTER_DAC14	21	DAC Threshold setting 14
REGISTER_DAC15	22	DAC Threshold setting 15
REGISTER_DAC16	23	DAC Threshold setting 16
REGISTER_DAC17	24	DAC Threshold setting 17
REGISTER_DAC18	25	DAC Threshold setting 18
REGISTER_DAC19	26	DAC Threshold setting 19
REGISTER_DAC20	27	DAC Threshold setting 20
REGISTER_DAC21	28	DAC Threshold setting 21
REGISTER_DAC22	29	DAC Threshold setting 22
REGISTER_DAC23	30	DAC Threshold setting 23
REGISTER_DAC24	31	DAC Threshold setting 24
REGISTER_DAC25	32	DAC Threshold setting 25
REGISTER_DAC26	33	DAC Threshold setting 26
REGISTER_DAC27	34	DAC Threshold setting 27
REGISTER_DAC28	35	DAC Threshold setting 28
REGISTER_DAC29	36	DAC Threshold setting 29
REGISTER_DAC30	37	DAC Threshold setting 30
REGISTER_DAC31	38	DAC Threshold setting 31
REGISTER_DAC32	39	DAC Threshold setting 32

All Registers contain one 8-bit binary value (1 byte), allowing for a range between 0 and 255 (decimal) or 0x00 and 0xFF (hexadecimal).

Register Data Definitions:

Register Name	Address	Description
REGISTER_ID_VERS	0	**Read-only register**. Writing to this register has no effect. High-order 4-bits contains *major version* number and low-order 4-bits the *minor version* number. The *RaDeKL_DEVICEINFO* structure returned by *RaDeKL_GetDeviceInfo* creates an appropriate *union* (overlay) to extract the major and minor version numbers.
REGISTER_DETECT	1	**Write-only register**. Setting bit 0 (0x01) initiates *single* detect, setting bit 2 (0x04) initiates *continuous* detect. Clearing all bits (0x00) stops *continuous* detection. The other bits are for MSSI internal use only.
REGISTER_TID_TIME	2	Interval for continuous detection. Values are 0 (1 second), 1 (500 ms), 2 (250 ms), 3 (100 ms) and 4 (50 ms).
REGISTER_RF_CONTROL	3	**Write-only register**. Setting bit 0 (0x01) causes the radar device to be reset to factory settings. The other bits are for MSSI internal use only.
REGISTER_TX_ATTEN	4	Sets the transmitter attenuation. Valid settings are in the range from 0 to 63 (decimal). For reference, a value of 63 causes 0 dB attenuation, 57 = -3 dB, 51 = -6 dB and 43 = -10 dB. See function definition of *RaDeKL_SetTransmitAttenuation* for more information.
REGISTER_RX_ATTEN	5	Sets the receiver attenuation. Valid settings are in the range from 0 to 255 (decimal). For reference, a value of 0 causes 0 dB attenuation, 91 = -10 dB and 157 = -20 dB. See function definition of *RaDeKL_SetReceiveAttenuation* for more information.
REGISTER_RANGE	6	Causes detection to be shifted by 512 range bin increments (256 feet at 0.5 foot resolution).
REGISTER_DELAY	7	Causes detection to be shifted by 8 range bin increments (4 feet at 0.5 foot resolution).
REGISTER_DAC1 thru **REGISTER_DAC32**	8 - 39	Contain the 32 DAC Threshold values.

RaDeKLAPI.H Header File Listing

```
//#############################################################################
//#############################################################################
//
// RaDeKL API.h
//
// (header file)
//
// The RaDeKL API provides access to the functions of the Multispectral
//  Solutions, Inc. (MSSI) RaDeKL Radar product. Please refer to the
//  "RaDeKL Radar API Programmer's Guide" for details.
//
//#############################################################################
//#############################################################################

// Include only ONCE
#pragma once


// Include Windows stuff
#include <windows.h>
#include <stdio.h>
#include <math.h>
#include <sys/timeb.h>

// Include the FTDI USB Chip API header
#include "FTD2XX.h"


//
//
// Constant declarations
//
//
// Max sizes for device serial number and description strings
#define SIZE_SERIAL_NUMBER              16
#define SIZE_DESCRIPTION                64

// This string must occur somewhere within the device DESCRIPTION string
#define RaDeKL_DESCRIPTION              "RaDeKL WBT Radar B"

// Status message codes (used in conjunction with the codes from FTD2XX.h)
// --- FTDI specicific status codes
#define RaDeKL_OK                           FT_OK                           // Currently 0
#define RaDeKL_INVALID_HANDLE               FT_INVALID_HANDLE               // Currently 1
#define RaDeKL_DEVICE_NOT_FOUND             FT_DEVICE_NOT_FOUND             // Currently 2
#define RaDeKL_DEVICE_NOT_OPENED            FT_DEVICE_NOT_OPENED            // Currently 3
#define RaDeKL_IO_ERROR                     FT_IO_ERROR                     // Currently 4
#define RaDeKL_INSUFFICIENT_RESOURCES       FT_INSUFFICIENT_RESOURCES       // Currently 5
#define RaDeKL_INVALID_PARAMETER            FT_INVALID_PARAMETER            // Currently 6
#define RaDeKL_INVALID_BAUD_RATE            FT_INVALID_BAUD_RATE            // Currently 7
#define RaDeKL_DEVICE_NOT_OPENED_FOR_ERASE  FT_DEVICE_NOT_OPENED_FOR_ERASE  // Currently 8
#define RaDeKL_DEVICE_NOT_OPENED_FOR_WRITE  FT_DEVICE_NOT_OPENED_FOR_WRITE  // Currently 9
#define RaDeKL_FAILED_TO_WRITE_DEVICE       FT_FAILED_TO_WRITE_DEVICE       // Currently 10
#define RaDeKL_EEPROM_READ_FAILED           FT_EEPROM_READ_FAILED           // Currently 11
#define RaDeKL_EEPROM_WRITE_FAILED          FT_EEPROM_WRITE_FAILED          // Currently 12
#define RaDeKL_EEPROM_ERASE_FAILED          FT_EEPROM_ERASE_FAILED          // Currently 13
#define RaDeKL_EEPROM_NOT_PRESENT           FT_EEPROM_NOT_PRESENT           // Currently 14
#define RaDeKL_EEPROM_NOT_PROGRAMMED        FT_EEPROM_NOT_PROGRAMMED        // Currently 15
#define RaDeKL_INVALID_ARGS                 FT_INVALID_ARGS                 // Currently 16
#define RaDeKL_NOT_SUPPORTED                FT_NOT_SUPPORTED                // Currently 17
#define RaDeKL_OTHER_ERROR                  FT_OTHER_ERROR                  // Currently 18

// --- RaDeKL API specific status codes
#define RaDeKL_READ_TIMEOUT                 201
#define RaDeKL_WRITE_TIMEOUT                202
#define RaDeKL_INCORRECT_SERIAL_NUMBER      203
#define RaDeKL_WRITE_REGISTER_FAILED        204
#define RaDeKL_READ_REGISTER_FAILED         205
#define RaDeKL_READ_DETECTION_FAILED        206
```

```
#define RaDeKL_BAD_THRESHOLD                207
#define RaDeKL_BAD_TX_ATTEN                 208
#define RaDeKL_BAD_RX_ATTEN                 209
#define RaDeKL_BAD_RANGE_DELAY              210
#define RaDeKL_BAD_INTERVAL                 211
#define RaDeKL_UNKNOWN_RESOLUTION           212

// Command codes
#define CMD_READ                        114
#define CMD_WRITE                       119
#define CMD_TERM                        255
#define CMD_DATA                        234

// Detection register codes
#define DETECT_SINGLE                   1
#define DETECT_START_CONTINUOUS         4
#define DETECT_STOP_CONTINUOUS          0

// RF-Control register codes
#define FIRMWARE_RESET                  1

// Resolution setting constants
#define RESOLUTION_UNKNOWN              0
#define RESOLUTION_6INCHES              1
#define RESOLUTION_1FOOT                2

// Continuous Collection Interval setting constants
#define INTERVAL_1_SEC                  0
#define INTERVAL_500_MS                 1
#define INTERVAL_250_MS                 2
#define INTERVAL_100_MS                 3
#define INTERVAL_50_MS                  4

// Version-Specific limits (see struct RaDeKL_DEVICEINFO below)
#define V1_RESOLUTION                   RESOLUTION_1FOOT
#define V1_RANGE_BINS                   256
#define V1_RANGE_BIN_MIN                0
#define V1_RANGE_BIN_MAX                32
#define V1_THRESHOLDS                   32
#define V1_THRESHOLD_MIN                20
#define V1_THRESHOLD_MAX                227
#define V1_TX_ATTEN_MIN                 0
#define V1_TX_ATTEN_MAX                 63
#define V1_RX_ATTEN_MIN                 0
#define V1_RX_ATTEN_MAX                 255
#define V1_INTERVAL_MIN                 0
#define V1_INTERVAL_MAX                 4

// RaDeKL Register addresses (16-bit)
#define REGISTER_ID_VERS            0   // Version ID register
#define REGISTER_DETECT                 1   // Detection command register
#define REGISTER_TID_TIME               2   // Interval for continuous detections (milliseconds)
#define REGISTER_RF_CONTROL             3   // RF Control register
#define REGISTER_TX_ATTEN               4   // Transmitter attenuation register
#define REGISTER_RX_ATTEN               5   // Receiver attenuation register
#define REGISTER_RANGE                  6   // RANGE register (shift by 512 range bins)
#define REGISTER_DELAY                  7   // DELAY register (shift by 8 range bins)
#define REGISTER_DAC1                   8   // Threshold setting 1
#define REGISTER_DAC2                   9   // Threshold setting 2
#define REGISTER_DAC3                   10  // Threshold setting 3
#define REGISTER_DAC4                   11  // Threshold setting 4
#define REGISTER_DAC5                   12  // Threshold setting 5
#define REGISTER_DAC6                   13  // Threshold setting 6
#define REGISTER_DAC7                   14  // Threshold setting 7
#define REGISTER_DAC8                   15  // Threshold setting 8
#define REGISTER_DAC9                   16  // Threshold setting 9
#define REGISTER_DAC10                  17  // Threshold setting 10
#define REGISTER_DAC11                  18  // Threshold setting 11
#define REGISTER_DAC12                  19  // Threshold setting 12
#define REGISTER_DAC13                  20  // Threshold setting 13
#define REGISTER_DAC14                  21  // Threshold setting 14
#define REGISTER_DAC15                  22  // Threshold setting 15
```

```
#define REGISTER_DAC16                  23  // Threshold setting 16
#define REGISTER_DAC17                  24  // Threshold setting 17
#define REGISTER_DAC18                  25  // Threshold setting 18
#define REGISTER_DAC19                  26  // Threshold setting 19
#define REGISTER_DAC20                  27  // Threshold setting 20
#define REGISTER_DAC21                  28  // Threshold setting 21
#define REGISTER_DAC22                  29  // Threshold setting 22
#define REGISTER_DAC23                  30  // Threshold setting 23
#define REGISTER_DAC24                  31  // Threshold setting 24
#define REGISTER_DAC25                  32  // Threshold setting 25
#define REGISTER_DAC26                  33  // Threshold setting 26
#define REGISTER_DAC27                  34  // Threshold setting 27
#define REGISTER_DAC28                  35  // Threshold setting 28
#define REGISTER_DAC29                  36  // Threshold setting 29
#define REGISTER_DAC30                  37  // Threshold setting 30
#define REGISTER_DAC31                  38  // Threshold setting 31
#define REGISTER_DAC32                  39  // Threshold setting 32
#define REGISTER_MAX                    39  // MUST be same as the last valid register


//
//
// Type definitions
//
//
// Handle to RaDeKL Radar device
typedef PVOID   RaDeKL_HANDLE;

// RaDeKL Radar device-specific info
typedef struct  RaDeKL_DEVICEINFO
{
    // FTDI USB Chip specific data
    FT_HANDLE   ft_handle;
    FT_DEVICE   ft_device_type;
    char        ft_serial_number[SIZE_SERIAL_NUMBER];
    char        ft_description[SIZE_DESCRIPTION];
    union       // This union allows us to overlay the 32-bit device_id
    {           //  with the 16-bit product_id and vendor_id
        DWORD   ft_device_id;
        struct
        {
            WORD    ft_product_id;  // Low word
            WORD    ft_vendor_id;   // High word
        };
    };

    // RaDeKL Radar specific data
    BYTE    resolution;     // Currently always RESOLUTION_1FOOT
    WORD    range_bins;     // Currently always 256
    BYTE    range_bin_min;  // Currently always 0
    BYTE    range_bin_max;  // Currently always 32
    BYTE    thresholds;     // Currently always 32
    BYTE    threshold_min;  // Currently always 20
    BYTE    threshold_max;  // Currently always 227
    BYTE    tx_atten_min;   // Currently always 0
    BYTE    tx_atten_max;   // Currently always 63
    BYTE    rx_atten_min;   // Currently always 0
    BYTE    rx_atten_max;   // Currently always 255
    BYTE    interval_min;   // Currently always 0
    BYTE    interval_max;   // Currently always 4
    BYTE    interval;       // Current interval setting

    union   // This union allows us to overlay the 8-bit version_id
    {       //  with the 4-bit major and 4-bit minor version_id
        BYTE    version_id;
        struct
        {
            unsigned int    version_id_minor:4; // Low nibble
            unsigned int    version_id_major:4; // High nibble
        };
    };
} RaDeKL_DEVICEINFO;
```

```
//
//
// Forward declarations
//
//
// Status reporting utilities
char *RaDeKL_GetStatusText (ULONG errcode);
ULONG RaDeKL_GetAPIVersion ();

// RaDeKL Radar management functions
ULONG RaDeKL_ListRadars (DWORD *cnt, char ***serial_numbers, char ***descriptions);
void  RaDeKL_ListRadarsCleanup (char **list);
ULONG RaDeKL_OpenRadar (RaDeKL_HANDLE *handle, char *serial_number);
ULONG RaDeKL_CloseRadar (RaDeKL_HANDLE handle);
ULONG RaDeKL_FlushIO (RaDeKL_HANDLE handle);
ULONG RaDeKL_ResetRadar (RaDeKL_HANDLE handle);
ULONG RaDeKL_GetDeviceInfo (RaDeKL_HANDLE handle, RaDeKL_DEVICEINFO *info);

// RaDeKL Radar parameter functions
ULONG RaDeKL_SetThresholds (RaDeKL_HANDLE handle, BYTE *thresholds);
ULONG RaDeKL_GetThresholds (RaDeKL_HANDLE handle, BYTE *thresholds);

ULONG RaDeKL_SetTransmitAttenuation (RaDeKL_HANDLE handle, BYTE  attenuation);
ULONG RaDeKL_GetTransmitAttenuation (RaDeKL_HANDLE handle, BYTE *attenuation);

ULONG RaDeKL_SetReceiveAttenuation (RaDeKL_HANDLE handle, BYTE  attenuation);
ULONG RaDeKL_GetReceiveAttenuation (RaDeKL_HANDLE handle, BYTE *attenuation);

ULONG RaDeKL_SetRangeDelay (RaDeKL_HANDLE handle, DWORD  delay_feet, DWORD *actual_delay_feet);
ULONG RaDeKL_GetRangeDelay (RaDeKL_HANDLE handle, DWORD *delay_feet);

ULONG RaDeKL_SetInterval (RaDeKL_HANDLE handle, BYTE  interval);
ULONG RaDeKL_GetInterval (RaDeKL_HANDLE handle, BYTE *interval);

bool  RaDeKL_SimulatorMode (bool enable);

// RaDeKL Radar detection functions
ULONG RaDeKL_DetectSingle (RaDeKL_HANDLE handle, BYTE *data);

ULONG RaDeKL_StartContinuousDetection (RaDeKL_HANDLE handle);
ULONG RaDeKL_StopContinuousDetection  (RaDeKL_HANDLE handle);
ULONG RaDeKL_ReadDetectionData (RaDeKL_HANDLE handle, BYTE *data);

// RaDeKL Radar register functions
ULONG RaDeKL_WriteCheckRegister (RaDeKL_HANDLE handle, WORD addr, BYTE  value);
ULONG RaDeKL_WriteRegister      (RaDeKL_HANDLE handle, WORD addr, BYTE  value);
ULONG RaDeKL_ReadRegister       (RaDeKL_HANDLE handle, WORD addr, BYTE count, BYTE *values);

// RaDeKL Radar low-level data I/O functions
ULONG RaDeKL_SendCommand     (RaDeKL_HANDLE handle, BYTE *cmd,      DWORD bytes_to_send);
ULONG RaDeKL_ReceiveResponse (RaDeKL_HANDLE handle, BYTE *response, DWORD bytes_requested,
                              DWORD *bytes_received);
```