Power Matters

Microsemi

VP-API-II

Getting Started

Introduction

- The purpose of this presentation is to provide a summary of the VP-API-II from the perspective of the Application Developer.

- After this presentation you will:
  - Have an understanding of VP-API-II and be able to create a simple Voice application using VP-API-II.
  - Understand how VP-API-II can be used in a wide range of Application architectures.

- This material is intended for:
  - Software Engineers designing applications using VP-API-II
  - Anyone interested in understanding VP-API-II

- It is assumed that the reader has a basic understanding of the C language.

Agenda

- VP-API-II Main Concepts
  - Goals of VP-API-II: Simplify Voice, Architecture Independent, OS Independent
  - Contexts and Objects
  - Profiles and Options
  - Hardware Abstraction Layer (HAL) and System Services

- Application Decisions
  - Memory Management
  - Device/Line Control and Query
  - Multi or Single Process

- Device Family Differences
- Creating a QuickStart Application

Goals of VP-API-II: Simplify Voice

- Abstract device and line configuration and control for voice applications:
  - Device and Line Configuration
    - PCLK/MCLK Interface
    - AC, DC, Ringing Parameters
  - Ring Entry/Exit Handling
    - Switch Hook Debounce for Ringing state changes
    - Automatic Ring Exit State on Switch Hook
- Add commonly needed functions to simplify the application:
  - Caller ID:
    - Type I requires a function call to setup the Ringing Cadence and Caller ID Cadence that meets the CID requirements, then another function call to “load” the CID message data into the API-II.
    - Type II requires a single function call to setup the Caller ID Cadence that meets the CID requirements and loads the CID message data.
  - Dial Pulse/Flash Hook Detection and Generation
  - Ringing and Tone Cadencing

Goals of VP-API-II: Architecture Independent

- VP-API-II contains no Memory
  - All memory required by the VP-API-II is provided by the Application and can be dynamically or statically allocated.
    - Note: If the application removes the memory for a line (line context), it must call the function VpFreeLineCtx() to prevent the API-II from accessing it.

- VP-API-II is re-enterant
  - Re-enterency is the ability to allow multiple applications to call the same function.
  - The VP-API-II will detect re-enterant conditions and will either handle the requested operation or throw an error.
  - Conditions that cause a re-enterency error:
    - Hardware Access to the same device by two or more applications at the same time.
    - Code critical sections where shared data is being modified.
    - Performing multiple operations on the same line at the same time.

Goals of VP-API-II: Non-OS specific

- VP-API-II code does not rely on any user or OS specific functionality. Only the System Services or HAL layers that are required by the API-II may be OS specific.
  - System Services are those functions common to many applications in a given software architecture.
  - HAL (Hardware Abstraction Layer) is an implementation of the hardware access for the customer specific platform. No assumptions are made by the API-II regarding the HAL layer.

- The VP-API-II has been tested in a non-OS environment, in Linux User space, and deployed as a Linux Kernel driver and in VxWorks.
  - See "API-II to Linux Porting Guide" (included with the API-II install) for recommendations regarding the use of the API-II in a Linux Kernel.

Contexts and Objects

- Contexts and Objects are fundamental to VP-API-II:
  - (Device and Line) Contexts: A generic type that is used to access a specific device or line object. These are called VpDevCtxType and VpLineCtxType respectively.
  - (Device and Line) Objects: A specific type that contains data for a device or line. Some examples as they are called in the VP-API-II:
    - VE880 Series – Vp880DeviceObjectType, Vp880LineObjectType
    - VE890 Series – Vp890DeviceObjectType, Vp890LineObjectType

- VP-API-II functions accept pointers to VpDevCtxType and VpLineCtxType for the following reasons:
  - The application never needs to change to support multiple devices since the type passed is generic (as opposed to passing the objects).
  - The VP-API-II accesses devices and lines associated with one another through the contexts passed (required for multi-threaded applications).
  - The VP-API-II modifies data in the objects, which is pointed to by the contexts.

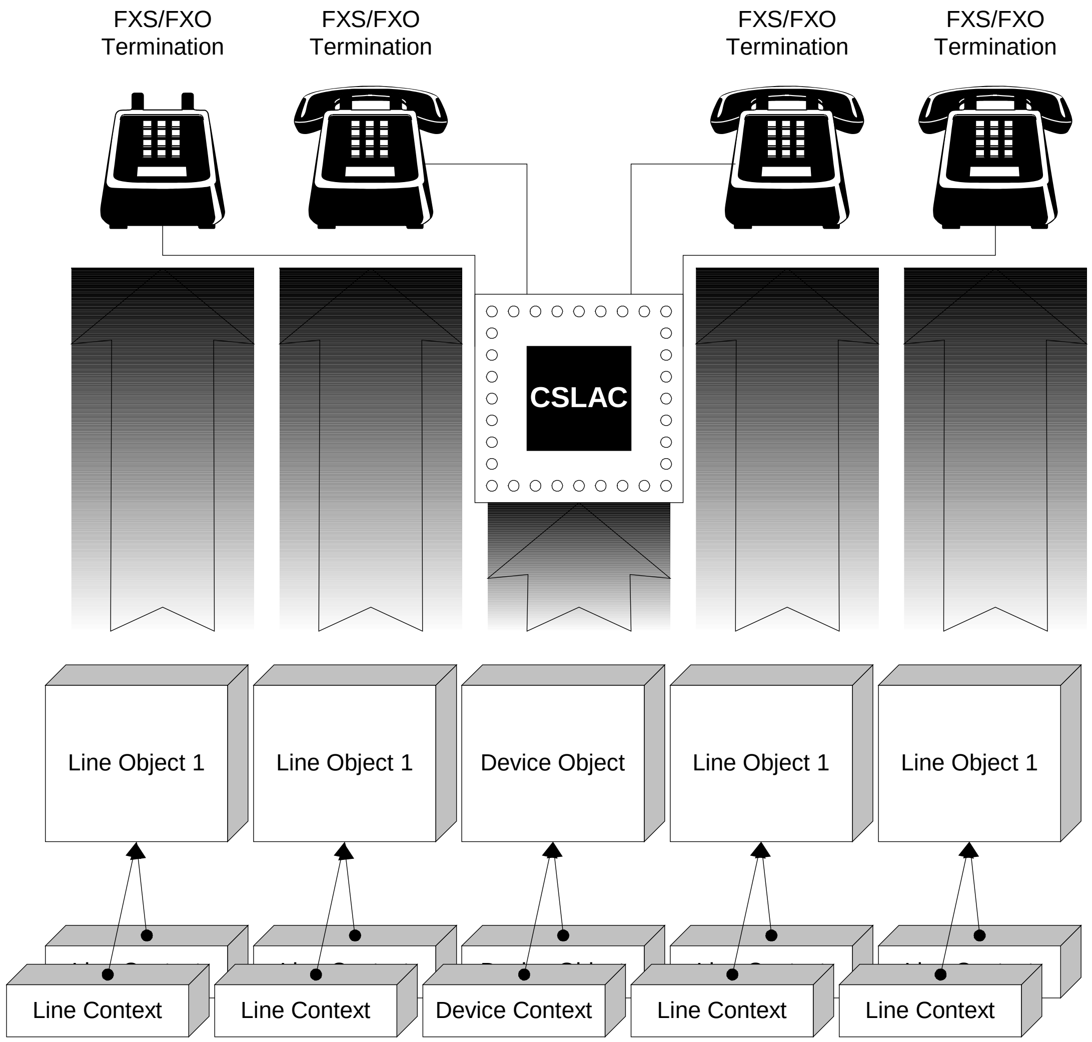
Contexts and Objects cont..

- The application creates all contexts and objects but passes only the contexts into the VP-API-II functions except …
  - .. in special API-II functions when a context and object are linked:
    - VpMakeDeviceObject() - Links a device context to a device object
    - VpMakeLineObject() – Links a line context to a line object

- Since contexts are generic and objects are specific, then each device/line can only have one object associated with it and must be shared among all applications accessing it, but each application can have it's own context for the same device/line.
  - The object data contains specific information regarding a device or line "at that time" and therefore must be common.
  - The context only provides a (generic) access to the object data.

- The figure on the next page is used to clarify the relationship between contexts, objects, and real devices/lines.

How are Device/Line Contexts and Objects related to real devices/lines?



One Device and Line Object for each Voice Termination Device and Line

Many Device and Line Contexts are possible for each Device and Line Object

Contexts and Object cont..

- The application may call every VP-API-II function with every parameter independent of the device/line type.
  - If the operation cannot be supported, the API-II function will return an error code. The specific error code will depend on the cause of the error.
  - If an error code is returned, no operation was performed on the specific device/line and no event is generated.

- Note: It is best write the application such that proper (or expected) error codes are returned in all places of the application.

Profiles and Options

- Profiles: Generally, items in the VP-API-II that an application will “set and forget”.
- Options: Generally, items that may change several times during program execution.

- Required Profiles for FXS only applications:
  - Device: Basic information to initialize and enable device communication to proceed.
  - AC: Filter coefficients designed to meet country specific requirements.
  - DC: Parameters affecting DC conditions on the line and loop supervision excluding those affected by ringing.
  - Ringing: Parameters of the Ringing signal (Amplitude, Frequency, Offset, etc.) and loop supervision parameters affecting ring trip operation.

- Useful Profiles for FXS applications
  - Caller ID: Defines Country specific line and PCM control, and message formating parameters, to generate Caller ID.
  - Ringing Cadence: Defines country specific ringing cadence timing and sequencing, including “when” to provide Caller ID.

Profiles and Options cont..

- Feature Specific Profiles: Profiles not required by a basic VP-API-II application, but simplify the application when the specific function is required.
  - Dialing
    - DTMF and Dial Pulse parameters used on an FXO line type when the FXO circuitry is dialing out.
  - Tone
    - Line level tones used for Call Progress (or test).
  - Cadence (Tone and Ringing)
    - Enables API-II automatic cadencing of Tones and Ringing on the line. Also used with Caller ID Profile to sequence Ringing with Caller ID.
  - Caller ID
    - Combined FSK (or DTMF) parameters and line state control used for Caller ID.
  - Metering
    - Defines Parameters used by the line when a Metering signal is generated.

Profiles and Options cont..

- Where are the profiles used?
  - Profiles required by VP-API-II are used in the functions required by a basic VP-API-II application:
    - VpInitDevice()
    - VpInitLine()
    - not required by the application if all line context and objects have been created (using VpMakeLineObject()) prior to calling VpInitDevice(). VpInitDevice() will initialize the device and all lines associated with the device.
    - VpConfigLine()
    - not required by the application. All initial line configuration can be done with VpInitDevice()
  - Profiles not required by a basic VP-API-II application are used in feature specific functions:
    - VpInitCid()
    - VpStartMetering()
    - VpSendTone()

Profiles and Options cont..

- Options are used to set every other device or line configuration parameter that is not set by Profiles. Every (reasonable) application will use options.

- Functions used to set and get options:
  - VpSetOption() – Sets options for a device, a line, or for all lines associated with a device. See the VP-API-II Reference Manual for details.
  - VpGetOption() – Retrieves option data for the device or line. Note that it is not possible to retrieve a line option for all lines at the same time.

- When retrieving options, the VP-API-II will generate an event indicating that the read option data is ready. The application needs to process the event and call VpGetResults() to access the option data.

Profiles and Options cont..

- The list provides some of the device and line options used by VP-API-II. This list is not intended to be inclusive. The reader should refer to the VP-API-II Reference Manual for a complete list and details.
  - (some) Device Specific Options:
    - VP_DEVICE_OPTION_ID_PULSE /* Specify pulse & flash decode timings */
    - VP_DEVICE_OPTION_ID_CRITICAL_FLT /* Specify action to take on critical fault */
    - VP_DEVICE_OPTION_ID_DEVICE_IO /* Specify Device IO options */
  - (some) Line Specific Options:
    - VP_OPTION_ID_ZERO_CROSS /* Select zero-cross ring-entry/exit options */
    - VP_OPTION_ID_PULSE_MODE /* Specifies pulse digit decode on or off. */
    - VP_OPTION_ID_TIMESLOT /* Specify Transmit and Receive Timeslots*/
    - VP_OPTION_ID_CODEC /* Specify PCM encoding */
    - VP_OPTION_ID_PCM_HWY /* Select the active PCM highway (A or B) */
    - VP_OPTION_ID_LOOPBACK /* Specify loopback mode */
    - VP_OPTION_ID_EVENT_MASK /* Specify the event mask (Enable/Disable specific events) */
    - VP_OPTION_ID_RING_CNTRL /* Specify the options for ringing control */

HAL and System Services

- HAL and System Services refers to a set of code specific to the platform (hardware and software/OS) that is required for the VP-API-II to work.

- The specific HAL and System Services functions to be implemented will depend on the API-II device family selected.
  - Refer to the chapter titled “Hardware Abstraction Layer” for HAL functions to be implemented.
  - Refer to the chapter titled “System Services” for system services functions to be implemented.
    - Note: Some system services functions are provided to support multi-thread architectures and may be empty if not required.

Application Decisions

- The purpose of this section is to explain how VP-API-II fits into the application based on a few of the application decisions that will be made. Note that the VP-API-II does not impose limitations on these decision.

- Common Application Decisions:
  - Memory Management
  - Device/Line Control and Query
  - Multi or Single Process

- The VP-API-II functions are specified to behave a specific way independent of the application architecture. However, some behavior can only occur in specific architectures and require the use of additional VP-API-II functions.

Application Decisions: Memory Management

- VP-API-II leaves all memory management decisions to the application. But there are three models to consider:
  - Model 1: Statically allocated, Single Process – All memory is allocated before initialization (VpInitDevice()) and never removed during execution. VP-API-II functions are called strictly sequentially.
  - Model 2: Dynamically allocated, Single Process – Memory is allocated/deallocated during program execution. VP-API-II functions are called strictly sequentially.
  - Model 3: Statically or Dynamically allocated, Multi-Process – Memory may be allocated/deallocated during program execution. VP-API-II functions may be called while the same (or other) VP-API-II functions are being executed.

- Statically allocated, Single Process is the simplest model and will be discussed first.

Application Decisions: Memory Management cont...

- Model 1: Statically allocated, Single Process
  - API-II does not encounter a re-enterency -- requested function always executes.
  - All contexts and objects used by the application (and VP-API-II) always exist

- The application will instantiate the contexts/objects as follows (assume one device and two lines):
  - VpDeviceCtxType devCtx;
  - VpLineCtxType lineCtx[2]; /* Assume Two lines */
  - VE880DeviceObjType ve880DevObj;
  - VE880LineObjType ve880LineObj[2]; /* Match num of ctx */

- The devices and lines are linked in VP-API-II as follows (assume device ID = 0):
  - VpMakeDeviceObject(&devCtx, &devObj[0], VP_VE880_SERIES, 0);
  - VpMakeLineObject(&lineCtx[0], &lineObj[0], &devCtx, VP_LINE_FXS_GENERIC, 0);
  - VpMakeLineObject(&lineCtx[1], &lineObj[1], &devCtx, VP_LINE_FXS_GENERIC, 1);

- VP-API-II Application Notes for Model 1: None. Call any VP-API-II function after this point.

Application Decisions: Memory Management cont...

- Model 2: Dynamically allocated, Single Process
  - API-II does not encounter a re-enterency -- requested function always executes.
  - Contexts and objects used by the application (and VP-API-II) do not always exist.

- The application will allocate/deallocate memory for the contexts/objects.

- The devices and lines are linked in VP-API-II the same way described for Model 1 (discussed previously).

Application Decisions: Memory Management cont...

- API-II Application Notes for Model 2:
  - If a device context or object is made invalid, the application must no longer call API-II functions for that device or any VP-API-II function for lines associated with that device.
  - If a line context or object is made invalid, the application needs to call VpFreeLineCtx(). This tells the VP-API-II that the line is no longer valid and should not try to access it's content.
  - Prior to discarding a line context or object, it is recommended that the line state first be set to Disconnect.
  - If a device context or object is discarded, it is recommended that all lines associated with that device first be set to Disconnect.

Application Decisions: Memory Management cont...

- Model 3: Statically or Dynamically allocated, Multi-Process
  - Code execution controlled by a scheduling algorithm and therefore VP-API-II functions are stopped/started at random points. The same function may be executing more than once.
  - Contexts and objects used by the application (and VP-API-II) may not always exist.

- Only one process can instantiate the device/line objects, all other processes will call a second set of special VP-API-II functions:
  - VpMakeDeviceCtx() – Links a device context to a device object without affecting the device object contents.
  - VpMakeLineCtx() – Links a line context to a line object without affecting the line object contents.

Application Decisions: Device/Line Control and Query

- (Basic) Application Notes:
  - CSLAC devices require a function call to VpApiTick() on a regular interval and contain software generated events (interrupts).
  - VpApiTick() is also used to implement sequences internal to VP-API-II (e.g., Ringing/Tone Cadence, Caller ID, Metering, etc.).
  - VpGetEvent() is used to determine if an event is active and if an event requires an additional result read.

- An application designed to work for all device families should call VpGetEvent() on a regular basis.

Application Decisions: Multi or Single Process

- There are no limitations in a single process application so they are not discussed further
- Multi-Process applications typically require re-enterancy.
- Devices, Lines, and MPI Bus are limited resources, so some rules are imposed by VP-API-II.
  - MPI access cannot be interrupted by other MPI access on the same device.
  - Two functions cannot execute simultaneously on the same line. The second function called in this scenario will report an error.
  - Device level operations that affect all lines cannot be performed if any of it's lines are in an active function.

  - Note: The VP-API-II implements these operations by calling the functions VpSysEnterCritical() and VpSysExitCritical() that the user application must implement. For single process applications (i.e., applications where it is impossible to violate these rules due to program flow) these functions can be empty.

Creating a QuickStart Application

- QuickStart(s) – The “Hello World” of VP-API-II
  - A simple application implementing the minimal amount of code necessary to perform a specific task using VP-API-II.

- Basic rules for a QuickStart application:
  - Memory Management: Statically (global) allocated is simplest to use, so QS uses this model.
    - QS creates the VpDevCtx and VpLineCtx instances. It also creates the device/line type specific object instances.
  - Single Process: Assume no other process is running. So it will be necessary to create the links to the device/line contexts and objects by using functions:
    - VpMakeDeviceObj()
    - VpMakeLineObj()
  - Event Handling: Can simply poll VpGetEvent() at regular intervals, independent of the device family.

Creating a QS Application cont...

- We'll implement a CSLAC (VE880 device type) QuickStart, so it is necessary to call VpApiTick() at regular intervals.
  - Note: VpApiTick() is typically called at 10mS intervals, so we'll use that. Other times are possible, with some functional limitations.

- The application requires two 'C' functions:
  - main()
  - Some function to call VpApiTick() on a regular basis. We'll use apiTickHandler().
    - The QuickStart example code provided with the API-II install is targeted for the UVB running Linux 2.4. The lines that are specific to the UVB and Linux will be omitted in this document.

- Remember: VpApiTick() indicates if an event is active.

Creating a QS Application cont…

- First, instantiate the device and line contexts and objects. The contexts are a generic VP-API-II type, the objects are device/line type specific:

```
VpDevCtxType devCtx;  /* Create generic device context */
VpLineCtxType lineCtx;  /* Create generic line context */
Vp880DevObjType vp880Dev; /* Create VP880 device specific object */
Vp880LineObjType vp880Line; /* Create VP880 line specific object */
```

  - Your application will refer to “devCtx, lineCtx, vp880Dev, and vp880Line”

- Begin implementing main()

```
main()
{
    int deviceId = 0;  /* Define the CS to be associated with the device */
    int chanId = 0;  /* Specify the channel # associated with the device */

    /* Remember: Must link the contexts to the objects */
    VpMakeDeviceObject(VP_DEV_880_SERIES, deviceId, &devCtx, &vp880Dev);
    VpMakeLineObject(VP_TERM_FXS_GENERIC, chanId, &lineCtx, &vp880Line, &devCtx);

    /* Note that ..MakeLineObject accepts the address of the device context for which it is associated. That is done to link
    specific lines to a specific device */
    /* From this point, the application will never refer to either “vp880Dev” or “vp880Line” again. All API-II functions are
    performed with the contexts */
```

Creating a QS Application cont…

- Continue implementing main()

```
main()
{
    …
    /* Context and Object initialization */
    …
    /* Now it is necessary to initialize the device and lines. The Profile Wizard creates the profiles data
       used in API-II. It is possible to initialize the lines without line specific profiles, although not
       recommended and the performance is unpredictable. But no additional API-II function will proceed
       without properly initializing the device, so a device profile is required. */
    /* For the purposes of this QS, well assume there is a device profile called “DEV_PROF”, and the line
       AC, DC, and Ringing profiles are called “AC600”, “DC25MA”, and “RING20HZ” respectively */

     VpInitDevice(&devCtx, DEV_PROF, AC600, DC25MA, RING20HZ, VP_PTABLE_NULL,
     VP_PTABLE_NULL);

     /* The last two values “VP_PTABLE_NULL” are places for FXO line specific profiles. Since this
     application is for an FXS line only, these entries are not needed. */
```

Creating a QS Application cont…

- Complete implementing main()

```
main()
{
    …
    /* Context and Object initialization */
    …
    /* Init Devices/Lines */
    …

    /* The details of implement the last section of main is platform specific. The application needs to call VpApiTick() at the
       interval specified in the device profile (typically 10mS), and if VpApiTick() indicates an active event, the application
       needs to process the event. */
    /* Add some operation here that implements calling a function, called “apiTickHandler()” every 10mS */

    /* Enter infinite loop so this application does not end. The apiTickHandler() is called every 10mS and is considered the
       “main” processing loop of the application from this point*/
    while(1);
    return 0; /* This code is never reached, but it keeps the compiler from generating warnings/errors */
}
```

Creating a QS Application cont…

- Implementing “apiTickHandler()”

```
apiTickHandler()
{
    /* The very first API-II function call should be VpApiTick() … that’s the point of this function!! VpApiTick() accepts the
        address of a boolean used to indicate if there is an active event */
    bool isEventActive = FALSE;  /* Declare and clear a boolean value to be used in VpApiTick() */
    VpEventType eventData;  /* If there is an event, we’ll need an instance of an event structure */

    VpApiTick(&devCtx, &isEventActive);  /* Make the 10mS call to VpApiTick() */

    /*Evaluate if there is an active event and if so, call VpGetEvent to access event information */
    if (isEventActive == TRUE) {  /* If “TRUE”, an event is active. The application must call VpGetEvent() */
        VpGetEvent(&devCtx, &eventData);
        /* eventData is filled with information describing a device level event. To determine if it is a line specific event, check
        the pLineCtx member of the event structure */
        if (eventData->pLineCtx != VP_NULL) { /* If the line context is valid, it is a line related event */
        /* Add code here to determine what the event category and Id is, then perform any line specific action based on the
            event */
         }
    }
    return 0;
}
```

- Congratulations! You just wrote your first VP-API-II Application.

Summary

- You now understand all of the major elements of the VP-API-II:
  - What is the difference between a "context" and "object"?
    - Contexts point to objects and contain non-specific information. Objects contain information that is specific to a particular device or line.
    - An application may have more than one context per device or line, but it can only have one object for each device/line.
  - Important functions:
    - VpMakeDevObject() – Links the device context to the device object and initializes the object contents.
    - VpMakeLineObject() – Links the line context to the line object and initializes the object contents.
    - EVERY APPLICATION MUST CALL THESE FUNCTIONS
  - How does the VP-API-II indicate device or line events (activity)?
    - Requires VpApiTick() to be called at regular intervals and indicates if an event is active.
    - Call VpGetEvent() to access the event data.

Summary cont…

- You also understand the minimum code necessary to create a QuickStart:
  1. Instantiate a device context and line context.
  2. Instantiate a device type specific device object and a line type specific line object.
  3. Call the functions: VpMakeDevObject() and VpMakeLineObject()
  4. Initialize the device and lines:
     1. Call VpInitDevice() with a minimum of a device profile and recommended with line specific AC, DC, Ringing, and FXO (if applicable) profiles.
  5. Call VpApiTick() at regular intervals
  6. Call VpGetEvent() to determine if an event is active.
  7. Based on the event that occurred, perform device/line actions using VP-API-II functions.

Summary cont…

- Lastly, you understand how VP-API-II fits into any software architecture:
  - VP-API-II contains no memory, all memory managed by the application.
  - VP-API-II supports re-enterency, which is the ability to call a given function more than one time without it exiting the first time.
  - VP-API-II supports multi-process applications by providing a mechanism to allow two independent process access to the same device/line information without interfering with each other.
  - VP-API-II can run either in an OS of the designers choice, or non-OS. It has been proven in Linux, VxWorks, and non-OS.

Where to Get More Information

- ‘C’ programming language references:
  - “C – A Reference Manual”, 5th edition, Harbison and Steele Jr.
  - “C Programming Language”, 2nd edition, Kernighan and Ritchie
- VP-API-II Reference Manual:
  - Describes all of the functions and structures used by the VP-API-II.
- QuickStart examples:
  - Source files included as part of the VP-API-II Install
    - The QuickStart applications are line module and “demo function” specific. Therefore it is recommended that you refer to the QuickStart source that most matches your application needs.
- For more information contact your local Microsemi FAE.