Verix eVo Multi-App Conductor

Programmers Guide

VeriFone Part Number DOC00306, Revision A

Verix eVo Multi-App Conductor Programmers Guide

Comments? Please e-mail all comments on this document to your local VeriFone Support Team.

VeriFone, Inc.
2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA
(800) VeriFone (837-4366)
www.verifone.com

VeriFone Part Number DOC00306, Revision A

CONTENTS

CHAPTER 3 Extended Event Services Layer

CHAPTER 4 Advanced EESL Usage

CHAPTER 5
Device Manager

CHAPTER 6
Resource Manager

CHAPTER 7
FrontEnd

Verix eVo Multi-App Conductor is called VMAC throughout the document. VMAC is an application manager designed to manage application selection, device usage, inter-task synchronization and communications, and many other multi-application environment related requirements within the Verix eVo platform.

Verix eVo Multi-App Conductor consists of a set of co-operating tasks and libraries, which help developers to design and develop applications. VMAC allows two or more VMAC-compliant tasks, that have been developed independently, to run together seamlessly on the same Verix eVo platform.

Target Audience

This document is intended for Verix eVo platform developers who use VMAC.

Scope

This document details the functionality of VMAC for Verix eVo family of terminals.

Document Organization

The manual is organized as follows:

Table 1 Document Organization

Chapter	Description
Chapter 1, Overview	Describes the features, terminology and functionality of VMAC.
Chapter 2, Libraries	Describes the usage of the VMAC shared library.
Chapter 3, Extended Event Services Layer	Describes the EESL, EESL events and the APIs.
Chapter 4, Advanced EESL Usage	Describes the advanced usage of EESL.
Chapter 5, Device Manager	Describes the Device Manager functionality, device mapping tables.
Chapter 6, Resource Manager	Describes the resource manager functionality.
Chapter 7, FrontEnd	Provides information on features, configuration, and customization options of the FrontEnd.
Chapter 8, Flexi-Records	Describes the concept of flexi-records and usage.
Chapter 9, LOGSYS Debug Utility	Explains how to debug applications written to run in the VMAC environment.
Chapter 10, Dynamic Menu	Describes the Dynamic Menu Library.
Chapter 11, XML Parser	Describes the XML Parser APIs exposed by the library.
Appendix A, Automatic Application Selection	Describes enabling automatic application selection.
Appendix B, VMAC Interface (VMACIF)	Describes the VMAC Interface (VMACIF) bridge application.

Conventions and Acronyms

The following conventions are used in this document.

- The `courier` typeface is used for code entries, filenames, and anything that might require typing at the DOS prompt or from the terminal keypad.
- The *italic* typeface indicates book title or emphasis.
- Text in blue indicates terms that are cross-referenced. When the pointer is placed over these references the pointer changes to the finger pointer, indicating a link. Click on the link to view the topic.

NOTE

Note points out interesting and useful information.

CAUTION

Caution points out potential programming problems.

Table 2 presents acronyms and their definitions.

Table 2 Acronym Definitions

Acronym	Definition
EESL	Extended Event Services Layer
GID	Group Identifier
IMM	Inter Task Message Manager
RM	Resource Manager

References

- Verix V Operating System Programmers Manual, VDN:23230
- Verix V Operating System Programming Tools Reference Manual, VDN:23231
- Verix eVo Multi-App Conductor for Touch screen Terminals Users Guide VDN: DOC00312

CHAPTER 1

Overview

VMAC allows two or more VMAC-compliant tasks, that have been developed independently, to run together seamlessly on the same Verix eVo platform. VMAC also allows any task to offer a service to other tasks on the same terminal.

An application is defined as being one or more executables, each with its own set of data files, and running as a separate task on a single Verix eVo terminal.

MAC and the Verix eVo Platform

VMAC runs on VeriFone Verix eVo terminals. The VMAC environment is based on the following core Verix eVo OS features:

- Multi Tasking capability: Verix eVo platform allows multiple tasks to run concurrently.
- Event based architecture: Each application (task) is driven by events received from the Verix eVo OS. These events are generated in response to a device activity such as a console key press or magnetic card swipe. An application may contain many tasks.
- Application Waterproofing: The Verix eVo platform provides hardware based waterproofing between tasks. The Verix eVo OS supports sixteen groups. The Verix eVo OS enforces special rules, as follows:
  - Group 0 (GID0) is the most privileged group. It is reserved for operating system files provided by VeriFone.
  - Group 1 (GID1) is the most privileged user group. It is intended for the primary application, typically belonging to the *owner* of the terminal. Another term often used is *sponsoring* application. This contains VMAC components, the shell application.
  - Groups 2 through 14 are intended for other applications.
  - Group 15 (GID15) can be used as a *universal* group, which will contain the shared libraries.
- The primary feature provided by the group ID is a private name space for file names. Within any particular file group all file names must be unique; but files from different groups are automatically distinct from one another. In practice, this gives each application a “private file space” in which all the tasks which are part of the application can see all of the application files, but tasks from outside the application can see none of them.
  Refer to the *Verix V Operating System Programmers Manual* for more information.

- Provision for inter-task communications using Pipes: Using the Verix eVo OS pipe implementation, it is possible to create a communication channel between any two applications (tasks) running on the terminal.
- Each device (such as Printer or Console) on the Verix eVo platform can only be *owned* by one application (task) at a time. The Verix eVo OS provides a mechanism to transfer device ownership between tasks.

Core VMAC Functionality vs VMAC Service Functionality

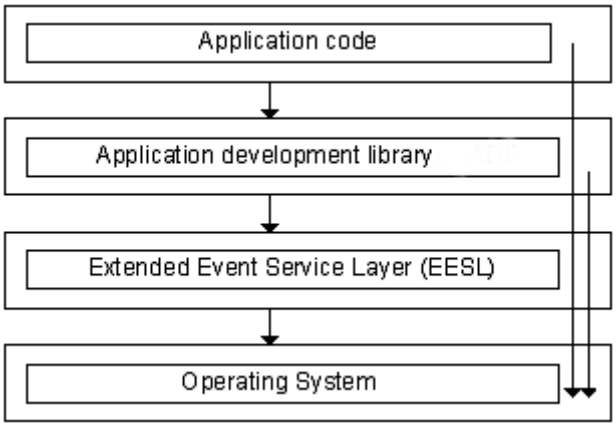
The core VMAC functionality ties the VMAC architecture together, and the VMAC service functionality is a scalable set of VMAC services that developers can include in their individual application development.

VMAC services exist as individual applications in their own right. They are built on top of the core VMAC architecture, and their sole purpose is to provide desirable VMAC service functionality to application developers.

The architecture is scalable because additional VMAC functionality can be added to a terminal by downloading an additional VMAC service application. The Verix eVo partial download functionality can be used to download the new VMAC service on its own. It is transparent to applications already on the terminal that a new VMAC service has been added.

Dependency Layers of a VMAC Necessary Application

The following diagram illustrates the library layers, and layer interdependency of a typical VMAC compliant application.



Understanding VMAC

Accessing core functionality and services

What does a task include to access VMAC core functionality and services?

To access the VMAC core functionality and VMAC services, a task must have access to a library that provides (for example, VMAC library) a set of interface functions that allows the task developers to access a broad range of VMAC functionality.

Inter-task Communication

How do tasks communicate using VMAC?

To make communication between co-operating tasks transparent, VMAC provides developers with the ability to send and receive Custom events. Each Custom event consists of a unique integer value, which identifies the event, and optional event data.

Custom events can be sent between any two tasks running in the terminal with VMAC, provided they have included the EESL layer mentioned above.

Creating your own custom events

How do developers create their own Custom events to send to other tasks?

Select a custom ID in the event range for the task. A maximum of 300 bytes of event data can accompany a Custom event if the task has registered using `EESL_Initialise()` function and a maximum of 1024 bytes if registered using `EESL_InitialiseEx()` function. The new Custom event, and event data, is easily sent to another task on the terminal using the `EESL_send_event` EESL API function.

Communication without connection

How can any two tasks communicate without establishing any connection?

All applications that run under VMAC, and want to use VMAC core functionality and services, must first register with VMAC. To register with VMAC a task makes a one off call to the EESL API function called EESL_Initialise() function or EESL_InitialiseEx() function. Registration with VMAC means that an indirect communications link has been made with all the other tasks registered with VMAC.

How does a task use functionality provided by VMAC Service Applications?

Each VMAC service application has a pre-defined set of *custom* events sent to it by other tasks to invoke specific service functionality. This could be described as a type of remote procedure call, where the function called is identified by the custom event value, and the function parameters are provided as custom event data. Service applications can also send custom events to other tasks. For example, a mail service application could send a custom event to a task to indicate it has a new mail message available.

How does device management tie in?

Sharing of devices (modem, printer, etc.) under VMAC is also established using custom events. The VMAC Device Manager sends custom events to a task to indicate that a specific terminal device is available or to request a device (or a set of devices), the task sends a custom event to the Device Manager. The Device Manager responds with a custom event that indicates the devices are available to the task or a custom event indicating the devices are already in use by other tasks and is not available.

Core Components

The following describes VMAC core components.

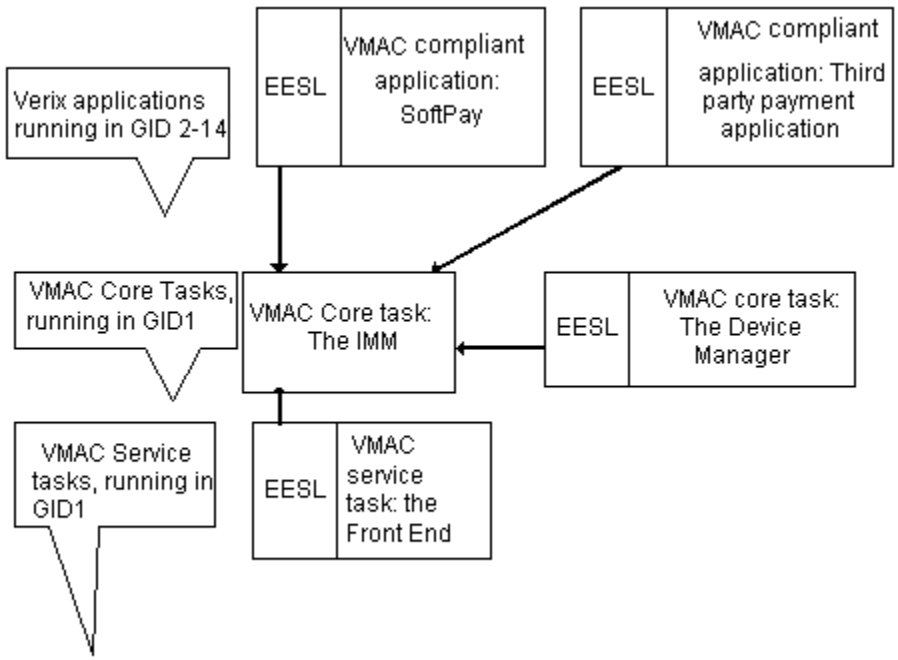
Inter Task Message Manager

This is provided in the form of a Verix eVo executable. The IMM is responsible for routing events between co-operating tasks. The IMM updates the `default.INI` file with the entries of the tasks in the individual `IMM*.INI` in each GID and starts all the tasks registered in the `default.INI` file. This task is transparent to all the tasks and the EESL layer works closely with IMM in passing events between tasks providing appropriate abstraction and encapsulation. This task runs in the background and is the core of VMAC architecture. The IMM provides mechanisms for inter task communication (using pipes of the Verix eVo OS). Applications and other tasks to be invoked during a cold power up, must register with IMM (using the API provided in EESL). IMM is responsible for invoking all the tasks and other VMAC components, for example, the Device Manager and FrontEnd.

Extended Event Services Layer

This layer is provided as a part of VMAC library and is linked with every executable running in the Multi Application Environment. This layer provides an API for a task to register with VMAC. This layer allows sending the tasks custom defined events (extended events) between tasks. The library is a layer above the OS that handles the set of extended events. This layer is called EESL. The communication of the extended events is dependent on the IMM utility which is a platform for intertask communication. The extended event functionality embeds events in IMM messages.

This layer provides APIs for sending, receiving events and data associated with it. The following diagram illustrates the interaction of the VMAC component set with other Verix eVo applications.



VMAC Service Applications

This section describes service applications that are available with VMAC.

FrontEnd

The FrontEnd component provides a menu for invoking applications manually. In addition, a screen saver is provided. If the FrontEnd fails to run a specified application, an appropriate error message displays.

Application developers can customize the FrontEnd for a deployment. Application developers must change the `screen saver.INI` file for any customizations to the screen saver. The FrontEnd is provided as a Verix eVo executable.

Refer to the Modifying the Screen Saver Display section for more details.

Device Manager

This is provided as a Verix eVo executable. This task is responsible for the allocation and de-allocation of devices assigned to the tasks running in the multi-application environment (for use with all VMAC compliant applications). All VMAC compliant tasks request ownership of the devices they require by sending a request to the Device Manager in the form of a custom event. If all the required devices are available, they are allocated to the application. The application is responsible for returning devices to the Device Manager after processing completes.

VMAC Environment

VMAC environment is defined as a terminal in which VMAC components are downloaded and installed in GID1. All other applications would run within the context of the VMAC environment. The VMAC components consist of the VMAC shared library (EESL, IMM, LOGSYS, Dynamic Menu), Device Manager and the FrontEnd.

VMAC Compliant Application

The following are the minimum changes required to be made to an existing stand alone application such that it can run in a VMAC environment.

To become VMAC compliant, all of the following must be performed:

1. An entry for the application is present in the `IMM*.INI` file downloaded to the same GID and drive as the `application.out` file.
2. The Application calls EESL_Initialise() function or EESL_InitialiseEx() function in the Application's main before any other initialization to register with the VMAC environment.
3. Provides an `AppName.INS` file in the same GID in which it is installed, where `AppName` is the application logical name.
4. Provides a device mapping table to handle activate event with all the devices required in the device list. This is the activate event listed in the `.INS` file.
5. When the application exits, on use of the HOTKEY (i.e., on `EVT_DEACTIVATE`) or by an exit menu item, the application closes all devices used and informs Device Manager that it has released all the devices, allocated earlier by the Device Manager.

6 The application requests the Device Manager for any device by sending event listed in the device mapping table. When the devices are no longer needed it closes all devices used and informs the Device Manager that it has released the devices.

Downloading VMAC

This section explains about the debug and release version of VMAC components. It also explains the procedure to download VMAC components and the samples provided as a part of VMAC media kit.

VMAC components are available in both debug and release version.

Debug version of VMAC facilitates viewing of debug statements using HyperTerminal that is connected to the COM port as defined by #LOGPORT `CONFIG.SYS` environment variable. Debug statements can be viewed only if LOGSYS flag is enabled. It can be enabled by changing `FRONTENDLOG` config variable, `DEVMANLOG` and `IMMLOG` value to C. It is recommended to use Debug version during development phase.

Release version of VMAC does not facilitate viewing of the debug statements. It is recommended to use release version during deployment phase.

<`EVOVMAC`> is the directory in which VMAC is installed.

Downloading VMAC Components

1 Go to the command prompt and change the directory to <`EVOVMAC`>`\output\RV\core\Download` for VX520 and <`EVOVMAC`>`\output\RV\core\Download\color` for VX680

2 Run the batch file `DlVmac` with the options as below:

`DlVmac d r` for downloading debug version to Ram

`DlVmac d f` for downloading debug version to Flash

`DlVmac r r` for downloading release version to Ram

`DlVmac r f` for downloading release version to Flash

This downloads a zip file which contains VMAC binaries and dld file(s) which contains all VMAC config variables. Refer to *Signing VMAC Binaries* section in *Readme* for more details on how to sign VMAC Binaries.

To disable screen saver in VX520, use the option `-S/-s`. By default, the screen saver is enabled.

Example:

`DlVmac r f -s` for downloading release version to Flash without screen saver.

NOTE

In VX520, if there are no applications downloaded the terminal displays "`No Applications Present`".

Downloading Samples

Go to the command prompt and change the directory to
`<EVOVMAC>\Samples\Download\RV`

Download the samples partially by running the `DlSample` batch file with the options as follows:

`DlSample d r` for downloading debug version to RAM

`DlSample d f` for downloading debug version to Flash

After the samples download to the terminal, a menu displays a list of all the application names.

NOTE

Even though VMAC library can run 8 applications in a GID, it is recommended to have one application per GID.

Libraries

VMAC contains the following libraries:

- **VMAC Library**
  - Dynamic Menu
  - Register Application
  - XML Parser
- **EESL Library**
  - EESL
  - Varrec
- **LSYS**
  - logsys

Libraries are available in both shared and static versions:

Libraries	Location

Debug version of shared libraries.	`<EVOVMAC>\output\RV\Lib\Files\Debug\vmac.lib`
	`<EVOVMAC>\output\RV\Lib\Files\Debug\eesl.lib`
	`<EVOVMAC>\output\RV\Lib\Files\Debug\lsys.lib`

Debug version of static libraries.	`<EVOVMAC>\output\RV\Lib\Files\Debug\vmac.a` `<EVOVMAC>\output\RV\Lib\Files\Debug\eesl.a`

Interface files for applications.	`<EVOVMAC>\output\RV\Lib\Files\Debug\vmac.o`
	`<EVOVMAC>\output\RV\Lib\Files\Debug\eesl.o`
	`<EVOVMAC>\output\RV\Lib\Files\Debug\lsys.o`

The release version of shared libraries are available at:

```
<EVOVMAC>\output\RV\Lib\Files\Release\vmac.lib
<EVOVMAC>\output\RV\Lib\Files\Release\eesl.lib
<EVOVMAC>\output\RV\Lib\Files\Release\lsys.lib
```

The release version of static libraries are available at:

```
<EVOVMAC>\output\RV\Lib\Files\Release\vmac.a
<EVOVMAC>\output\RV\Lib\Files\Release\eesl.a
```

The interface files for applications are available at:

```
<EVOVMAC>\output\RV\Lib\Files\Release\vmac.o
<EVOVMAC>\output\RV\Lib\Files\Release\eesl.o
<EVOVMAC>\output\RV\Lib\Files\Release\lsys.o
```

The header files for the VMAC library are:

```
<EVOVMAC>\include\DynamicMenu.h
<EVOVMAC>\include\RegisterApplication.h
<EVOVMAC>\include\Eeslapi.h
<EVOVMAC>\include\Varrec.h
<EVOVMAC>\include\VMAC.h
<EVOVMAC>\include\Logsys.h
<EVOVMAC>\include\Version.h
<EVOVMAC>\include\xmlparser.h
```

VMAC Library APIs

Following are the list of APIs provided by VMAC and the following APIs can be used to get the version of VMAC library:

- `GetVMACLibVersion()`

  This API returns the version of VMAC Library.

Prototype

```
#include <version.h>

char* GetVMACLibVersion(void);
```

Return Values

The version of the installed VMAC library.

Resource Manager

Resource Manager APIs are explained in the Resource Manager chapter.

Prototypes for these APIs are defined in `<EVOVMAC>\include\k2api.h`.

- `RM_CloseFile()`
- `RM_GetCurrNumRes()`
- `RM_GetCurrNumResOnType()`
- `RM_GetCurrResOnTypeId()`

- RM_GetCurrResOnTypeIndex()
- RM_GetCurrResTypeIdOnIndex()
- RM_GetFNameFromHandle()
- RM_GetNumRes()
- RM_GetNumResInFile()
- RM_GetNumResOnType()
- RM_GetPosFromHandle()
- RM_GetResIDFromHandle()
- RM_GetResInfo()
- RM_GetResOnTypeId()
- RM_GetResOnTypeIndex()
- RM_GetResSzFromHandle()
- RM_GetResTypeIdOnIndex()
- RM_LoadPartialRes()
- RM_LoadRes()
- RM_LoadResFromFile()
- RM_OpenFile()
- RM_ReplacePartialRes()
- RM_ReplaceRes()
- RM_ShutDown()
- RM_Start()

Flexi-Record APIs

Flexi-record APIs are explained in the Flexi-Records chapter.

Prototypes for these APIs are defined in `<EVOVMAC>\include\varrec.h`.

- vVarInitRecord()
- shVarAddField()
- shVarGetField()
- shGetRecordLength()
- shVarDeleteField()
- shVarQueryField()
- shVarUpdateField()
- ushInitStandardFlexi()
- shVarAddUnsignedInt()
- shVarGetUnsignedInt()
- shVarGetUnsignedChar()

- `shVarAddUnsignedChar()`
- `shVarAddUnsignedLong()`
- `shVarGetUnsignedLong()`

Dynamic Menu APIs

Dynamic Menu APIs are explained in the Dynamic Menu chapter.

Prototypes for these APIs are defined in `<EVOVMAC>\include\DynamicMenu.h`.

- `Add_Menu_Item()`
- `Remove_Menu_Item()`

Register Application API

The application can register with VMAC at runtime using this API.

Prototype

```
int registerApplication(char * name, unsigned char*
EESLglobalDataSpace, int size);
```

Parameters

`name`	Is the application logical name.
`eeslGlobalDataSpace`	Is the application global data used by EESL of size `EESL_DATA_SIZE_EX`.
`Size`	`EESL_DATA_SIZE_EX`

Return Values

Value	Description
INVALID_REGISTRATION_PARAMETERS	If the parameters passed are invalid.
RESOURCE_MANAGER_ERROR	Error in resource manager.
RESOURCE_FILE_NOT_FOUND	If the `<<APPLICATION_LOGICAL_NAME>>.res` file is not present.
MEMORY_ALLOCATION_ERROR	IF failed to allocate the required memory.
FILE_READ_ERROR	If not able to read the `<<APPLICATION_LOGICAL_NAME>>.ins` file.
INS_FILE_NOT_FOUND	If the `<<APPLICATION_LOGICAL_NAME>>.ins` file not found.

XML Parser APIs

XML Parser APIs are explained in XML Parser chapter. Prototypes for these APIs are defined in `<EVOVMAC>\include\XMLParser.h`.

Following are the available APIs:

- `ParseXML()`
- `ReadXMLFile()`
- `GetXMLElement()`
- `GetXMLAttribute()`
- `DeleteXMLElement()`
- `FindXMLElement()`
- `DeleteXMLRoot()`
- `CreateXMLStringR()`
- `SetXMLElement()`
- `WriteXMLFile()`

EESL APIs

EESL APIs are explained in the Extended Event Services Layer chapter.

Prototypes for these APIs are defined in `<EVOVMAC>\include\Eeslapi.h`.

- `EESL_Initialise()`
- `EESL_InitialiseEx()`
- `EESL_send_event()`
- `EESL_send_devman_event()`
- `EESL_send_res_release_event()`
- `EESL_send_res_release_eventEx()`
- `wait_event(), read_event(), wait_evt(), read_evt()`
- `EESL_read_cust_evt()`
- `EESL_read_evt_of_type()`
- `EESL_get_event()`
- `EESL_queue_count()`
- `EESL_appl_busy()`
- `EESL_appl_ready()`
- `EESL_is_appl_busy()`
- `EESL_check_app_present()`
- `EESL_get_outgoing_flexi()`
- `EESL_get_incoming_flexi()`
- `EESL_logical_name_list()`
- `EESL_get_task_list()`

- `EESL_get_number_of_tasks()`
- `EESL_set_send_retries()`

LOGSYS APIs

LOGSYS APIs are explained in the LOGSYS Debug Utility chapter.

Prototypes for these APIs are defined in `<EVOVMAC>\include\Logsys.h`.

- `LOG_INIT()`
- `LOG_PRINTF(())`
- `LOG_PRINTFF(())`
- `LOG_NONZERO_ERROR()`
- `LOG_ZERO_ERROR()`
- `LOG_NEGATIVE_ERROR()`
- `LOG_NONZERO_ERROR_CRIT()`
- `LOG_NEGATIVE_ERROR_CRIT()`
- `LOG_ZERO_ERROR_CRIT()`
- `LOG_NULL_POINTER()`

Downloading VMAC Shared Library

Follow the steps given below to download VMAC shared library:

1. Go to the command prompt and change the directory to `<EVOVMAC>\output\RV\lib\Download`
2. Run the `DlVmacLibf` batch file for downloading the VMAC shared library to GID15 in flash with the options as below:

   `DlVmacLibf d 1` for downloading debug version of shared library 1.

   `DlVmacLibf r 1` for downloading release version of shared library 1.
3. Run the `DlVmacLibr` batch file for downloading the VMAC shared library to GID15 in RAM with the options as below:

   `DlVmacLibr d 1` for downloading debug version of shared library 1.

   `DlVmacLibr r 1` for downloading release version of shared library 1.

CHAPTER 3

Extended Event Services Layer

The Verix eVo architecture is event based. The Verix eVo OS sends device events to all active tasks. There is no mechanism for sending application-defined events between co-operating tasks in the Verix eVo OS. The EESL is a layer above the OS that provides this functionary. EESL provides an API that allows applications to register with VMAC. Applications can define events and send them to another application running under VMAC. They are called *custom* events. EESL allows sending the custom defined events between applications. For the EESL to function, an executable, called `IMM.OUT`, is required. The IMM works in conjunction with the EESL library to switch custom events between applications. The EESL's use of the IMM is transparent to VMAC developers.

Custom Application Events

Custom application events are integer event values that are selected by developers to use in a specific application.

For example, an event `MAIL_AVAIL_EVT = 15000` can be defined and published by a mail service application. The mail application can send this event to other applications to indicate a new mail message for these applications.

Custom Event Range

Applications must define their own custom event range of integer values. The custom event set is not a subset or a superset of the Verix eVo operating system event set. Hence they are handled separately. The custom event set is integer values ranging from 10001 to 20000 and 20501 to 32767 (events 20001 to 20500 are claimed by the Device Manager).

Event Data

Custom events also differ from Verix eVo operating system events in that they can be assigned event data. If assigned, the event data is sent with the Custom event to the target application. Custom event data is limited to a maximum of 300 bytes if the application has registered with EESL_Initialise() function and it is limited to 1024 bytes if registered with EESL_InitialiseEx() function, but should be kept to a minimum if large event volume is expected from a particular custom event.

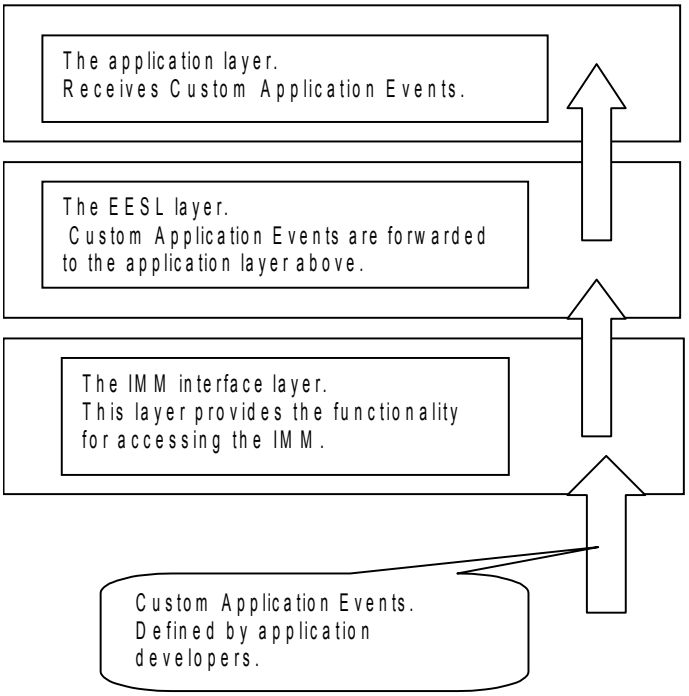
Custom Event Classes

Custom events are divided into `CLASS - A` and `CLASS - B` events.

The division of custom events into two separate classes has to do with application synchronization. In some cases, an application sending an event would like to know that the custom event is processed immediately. In other cases, the time when the receiving application processes the event may not be critical.

Event Type	Range	Description	Example
CLASS - A	10001–29999	These custom events must be processed immediately when received by an application.	• Device management events • Application selection events.
CLASS - B	30000–32767	These custom events can be processed at any time.	• Broadcast events. • Alert events, such as `MAIL_AVAILABLE`.

Flow of Custom Events



Intertask Message Manager

IMM is essentially the kernel of the VMAC architecture. The IMM has two purposes:

- It is responsible for running all applications (executables) under the VMAC environment. When the terminal is powered on, the IMM reads a configuration file for all the applications it is required to run. The IMM is the first task started by the Verix eVo terminal, i.e., `*GO=f:IMM.OUT`.
- IMM is used to switch Custom events to a destination application. This is transparent to any application developer using the EESL.

Configuring VMAC to Start an Application

In the VMAC architecture, all applications are launched by the `IMM.OUT` executable. VMAC is configured to launch an application by adding an entry to the `default.INI` text file. An entry in `default.INI` specifies the name of the executable to run and the logical name of the application. `default.INI` file is downloaded to GID1 in RAM.

Default.INI file

The default IMM INI file is provided by the #IMMINI config variable. The INI file downloaded using dlvmac is `IMM_VMAC.INI`.

Structure

- Within the INI text file, any text enclosed in square brackets is called a section.
- Each section identifies a unique set of keys.
- Each key is associated with a text or numeric value.
- Sections and keys are not case sensitive.
- Any data after ";" is ignored and can be used for comments in the `.INI` file.
- Blank lines are ignored. A line of text in the `default.INI` is limited to 128 characters. Lines longer than 128 characters are truncated to the first 128 characters.

The `default.INI` file has one defined section [Tasks].

[Tasks] Section

The [Tasks] section in the `default.INI` file describes which applications (each `.OUT` file) are run by VMAC and where these executables are located (the GID number, and flash or RAM directory). If the tasks to run are not registered with VMAC, use the '@' character.

Each task that has to be run by IMM is defined as a separate key under [Tasks]. The format is as follows:

```
key=file path
```

For example:

```
PRINTSP=2/F:PSM_MAIN.OUT; Printer/Spooler daemon in GID=2, FLASH
```

In the above example, the key is `PRINTSP` and the file path is

```
"2/F:PSM_MAIN.OUT"
```

The file path has the following format:

```
{GID number}/{flash(F) or RAM(I)}:{executable file name}
```

For example:

```
5/F:<application.out>
```

Each separator is used as follows:

Colon - The memory type where the application is resident precedes the colon. F represents the memory type for Flash, and I for RAM.

Forward slash - The GID number precedes the forward slash.

The file path can be specified indirectly via an environment variable in GID1 `CONFIG.SYS`.

For example:

```
PRINTSP=#CONFSPOOL
```

The # indicates that the path is specified by a CONFIG.SYS variable. The following entry appears in GID1 of CONFIG.SYS:

```
#CONFSPOOL =2/F:PSM_MAIN.OUT
```

or

```
#CONFSPOOL = PSM_MAIN.OUT
```

as path information is optional.

NOTE

- Each key in the tasks section is also to identify any application running under the VMAC environment. This key is also referred to as the application logical name. Whenever an application is addressed under VMAC its logical application name is used. Logical application names must be 2 to 14 characters in length.
- Adding path information as part of file path is optional.

Example Default.INI file

```
;
; tasks to run on power-on
;
[Tasks]
IDLE=1/F:IDLE.OUT; Idle application
PS=4/PSM_MAIN.OUT; Printer/Spooler daemon
KD=F:KD_MAIN.OUT; Console daemon in GID 1
PP=#PPDMN
;Pinpad daemon set in CONFIG.SYS #PPDMN variable
```

Adding an entry to [Tasks] section of default.INI

IMM of VMAC appends an entry consisting of the logical name and the location of a task to default.INI if IMM.INI file is present in the local GID in the task. The IMM.INI downloaded with the task to the local GID of the task can contain multiple entries in the following format:

```
{Application logical name}={GID number}/{Flash(F) or
Ram(I)}:{Executable file name}
OR
{Application logical name}={Executable file name}
```

For example:

```
PRINTSP=2/F:PSM_MAIN.OUT; Printer/Spooler daemon in GID=2, FLASH
OR
PRINTSP=PSM_MAIN.OUT; Printer/Spooler daemon in GID=2, FLASH
```

Default.INI vs IMM.INI

By default, IMM looks for an initialization file with the name as specified by the #IMMINI GID1 environment variable. This file contains all the required configuration keys as required by the IMM executable. The default.INI file must have the structure as mentioned.

```
; tasks to run on power-on
[Tasks]
```

```
DEVMAN=devman.out; Device Manager application
FRONTEND=frontend.out; Front End application
```

The tasks section can have all the tasks listed or provide the information in `IMM*.INI` files downloaded to the GID and drive of the tasks. The information provided in `IMM*.INI` overrides the entry already present for that logical name in `default.INI` file. If the entry for a task is present in the tasks section of the `default.INI` file, the task can be downloaded without `IMM.INI` file. If the entry for a task is not present in the `default.INI` file, then it is mandatory that the file `IMM.INI` is present in the GID where the task is present for that task to be run by IMM.

If the entry for a task is present in the tasks section of `default.INI` file, and a `IMM.INI` file is also present in the task's GID, the information in the `IMM*.INI` is used.

For example, assume that an application with logical name `APP1` is present in GID2 of the terminal and `APP2` in GID3 of Flash.

This information can be represented in three different methods.

- The tasks section in the `default.INI` file has the individual task entries apart from the mandatory entries:

  ```
  [Tasks]
  DEVMAN=devman.out (Device Manager application)
  FRONTEND=frontend.out (Front End application)
  APP1=2/Appln1.out
  APP2=3/f:Appln2.out
  ```

  And the tasks `APP1` and `APP2` can be downloaded without `IMM*.INI` files.

- The tasks section in the `default.INI` file has only the mandatory entries and the tasks provide `IMM*.INI` files. The ini file provided by the task must have the name `IMM*.INI` (can be `IMM.INI` also) and it can be only 21 characters long `(imm<APP_LOGICAL_NAME>.ini = 3+EESL_APP_LOGICAL_NAME_SIZE(14)+4).`

  ```
  [Tasks]
  DEVMAN=f:devman.out;(Device Manager application)
  FRONTEND=f:frontend.out;(Front End application)
  ```

NOTE

The name `default.INI` can be overridden by the `#IMMINI` configuration variable in the GID1 of the RAM. However this name cannot be `IIM.INI` as it reserved for the individual tasks and GID1 will have the initialization file information as follows.

`Immapp1.ini` file in GID2 has the following entry:

```
APP1=2/Appln1.out
```

or

```
APP1=Appln1.out
```

Immapp2.ini file in GID3 of flash has the following entry:

```
APP2=3/f:Appln2.out
```

or

```
APP2=Appln2.out
```

In the individual IMM*.INI files the path information (GID and drive information) is optional. The entries from the IMM*.INI files is added to the default.INI file. Following are the final list of tasks in default.INI file:

```
[Tasks]
DEVMAN=devman.out (Device Manager application)
FRONTEND=frontend.out (Front End application)
APP1=2/Appln1.out
APP1=3/f:Appln2.out
```

- The tasks section in the default.INI file has the individual task entries apart from the mandatory entries and the tasks also provide IMM*.INI files:

```
[Tasks]
DEVMAN=devman.out; Device Manager application
FRONTEND=frontend.out; Front End application
APP1=2/Appln1.out
APP2=3/f:Appln2.out
```

Immapp1.ini file in GID2 has the following entry:

```
APP1=2/Appln1.out
```

or

```
APP1=Appln1.out
```

Immapp2.ini file in GID3 of Flash will have the following entry

```
APP2=3/f:Appln2.out
```

or

```
APP2=Appln2.out
```

The entry in the default.INI file for APP1 is replaced with the entry from the immapp1.ini in GID2 and for APP2 the entry is replaced with the entry from immapp2.ini in GID3 of flash. The final list of tasks in default.INI is:

```
[Tasks]
DEVMAN=devman.out (Device Manager application)
FRONTEND=frontend.out (Front End application)
APP1=2/Appln1.out
APP2=3/f:Appln2.out
```

NOTE

- The initialization file of the application must have its name in the form of `IMM*.INI` and the filename should not be more than 19 characters long, including the extension and the path info. For example, `f:immapplogname.ini`. This ini file *must* be at the same location (same GID and drive) as the `.out` of the task.
- The entry in the `IMM*.INI` file provided by the task may or may not contain the path (GID and drive) information. IMM builds the path information from the location of the INI file and this overrides the path info provided in the INI file.

Application Arguments

Arguments can be passed to applications that run on the Verix eVo platform. These arguments are received in argc, argv in the application `main()`.

You can configure VMAC to pass these arguments when it starts an application listed in the `default.INI`, by defining the arguments in the `CONFIG.SYS` of the applications local GID.

The `CONFIG.SYS` variable has the following format:

```
{Application Logical Name}AR
```

The application arguments specified must contain only the argv values. The OS takes care of sending the argc to the application `main()` function.

For example, assume that there is the following `default.INI` entry:

```
[Tasks]
PRINTSP=PSM_MAIN.OUT; Printer/Spooler daemon
Set the following in config.sys to pass arguments I10 P20 to the
application:
PRINTSPAR = I10 P20
```

Using the EESL Library

`EESLAPI.H`, the `EESL` include file, must be inserted directly after standard C includes, and Verix eVo OS includes. For example, the following shows the position of `EESLAPI.H` after the C includes, and Verix eVo OS include i.e., `SVC.H`:

```
#include <stdio.h>
#include <string.h>
#include <svc.h>

/* EESL include must appear here */
/*********************************/
#include "eeslapi.h"

/* Other application includes start here */
```

VMAC Registration

Every task must register with VMAC before it can use the VMAC functionality.

After an application listed in the `default.INI` starts, VMAC waits for the task to register before running the next task in the `default.INI` list. This excludes tasks that have been configured not to register. If the task does not register within 5 seconds VMAC continues with the next application in the `default.INI`. This is the default time. To avoid this, the time out can be configured as:

The task registration time out can be changed by adding `PipeInpTimeout` variable with a corresponding value assigned in the `imm_vamc.ini` file, provided this variable is declared before declaring VMAC components.

E.g.,

```
[IMM]
PipeInpTimeout=5000

[Tasks]
LOGSVR=1/f:LOGSVR.OUT
```

The application must register soon after it has been run, preferably at the beginning of the application's `main()` argument.

`EESL` provides a simple function to encapsulate the registration process. This function must reside in the tasks `main()` function.

Registration Functions

The `EESL` provides a simple function that encapsulates the registration process. This function should reside in the application's `main()` function.

The following is the list of APIs provided by VMAC.

EESL_Initialise()

Prototype

```
long EESL_Initialise(char * argv[], short argc, char *
pchLogicalTaskName,EESL_CONTROL_STR * EESLglobalDataSpace);
```

Description

Call this function to register an application with the VMAC environment. If registered with this function the maximum event data allowed is 300 bytes.

Parameters

`argv`	Registers an application with the VMAC environment. If registered with this function the maximum event data allowed is 300 bytes.
`argc`	Contains the number of arguments passed to an application as received through the application `main()` function.
`pchLogicalTaskName`	A string containing the application logical name (refer to the Configuring VMAC to Start an Application section for more details). The application logical name is specified here, as well as in the `default.INI` as a precaution against an invalid logical name appearing in the `default.INI`. The string should only be in uppercase.

`EESLglobalDataSpace`	The `EESL` does not allocate its own global data space. This must be allocated by the application.

Return Values

Returns the following non-zero values if an error occurs:

Error Code	Description
EESL_ERROR_REG_FAILURE	VMAC is present, but registration process failed. This is a critical error.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.
EESL_INVALID_PARAMETER	Invalid parameter was passed to the function check for NULL pointers.

The following example shows the registration function in an applications `main()` function:

```
EESL_CONTROL_STR EESLGlobalData;
void main(int argc, const char* argv[])
{
/* FIRST STEP - Initialise EESL.SVR is the logical application name
specified in Default.INI */
error = EESL_Initialise(argv,argc,"SVR", &EESLGlobalData);
…..
…..
} // main
```

EESL_InitialiseEx()

Registers an application in the VMAC environment. The application can register with function for event data limit of 300 bytes or 1024 bytes.

If event data is within 300 bytes limit, call this method with `EESL_DATA_SIZE` function.

```
unsigned char eeslDataSpace[EESL_DATA_SIZE];
EESL_InitialiseEx(argv, argc, appLogicalName, eeslDataSpace,
                  EESL_DATA_SIZE);
```

If higher event data limit is required, (within 1024 bytes) call this method with `EESL_DATA_SIZE_EX`.

```
unsigned char eeslDataSpace[EESL_DATA_SIZE_EX];
EESL_InitialiseEx(argv, argc, appLogicalName, eeslDataSpace,
                   EESL_DATA_SIZE_EX);
```

The memory required must be provided by the application.

Prototype

```
long EESL_InitialiseEx(char * argv[], short argc, char *
                       pchLogicalTaskName,unsigned char *
                       EESLglobalDataSpace, short size);
```

Parameters

`argv`	This is argument array received through the `main()` application. VMAC passes connection information to an application when it starts the application.
`argc`	This is variable containing the number of arguments passed to an application as received through the `main()` application.
`LogicalTaskName`	This is a string containing the application logical name (refer to the Configuring VMAC to Start an Application section for more details). The application logical name is specified here, as well as in the `default.INI` as a precaution against an invalid Logical Name appearing in the `default.INI`.
`EESLglobalDataSpace`	The EESL does not allocate its own global data space. This must be allocated by the application. The application can allocate this space by declaring a global variable of type unsigned char * of the size `EESL_DATA_SIZE` or `EESL_DATA_SIZE_EX` depending on the size of event data required.

Return Values Returns the following non-zero values if an error occurs:

Error Code	Description
EESL_ERROR_REG_FAILURE	VMAC is present, but registration process failed. This is a critical error.

EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to `EESL API` functions following this will return the same error.
	The following example shows the registration function in an application's `main()` function:
	`unsigned char EESLGlobalData[EESL_DATA_SIZE_EX};`
	`void main(int argc, const char* argv[])`
	`{`
	`/* FIRST STEP - Initialise EESL.SVR is the logical application name specified in Default.INI */`
	`error = EESL_InitialiseEx(argv,argc,"SVR", EESLGlobalData, EESL_DATA_SIZE_EX);`
	`..`
	`..`
	`} // main`
	or
	`unsigned char EESLGlobalData[EESL_DATA_SIZE];`
	`void main(int argc, const char* argv[])`
	`{`
	`/* FIRST STEP - Initialise EESL.SVR is the logical application name specified in Default.INI */`
	`error = EESL_InitialiseEx(argv,argc,"SVR", EESLGlobalData, EESL_DATA_SIZE);`
	`..`
	`..`
	`} // main`
	The second example is same as calling EESL_Initialise() function, in that the custom event data size is limited to 300 bytes.

EESL_INVALID_PARAMETER	Invalid parameter was passed to the function check for NULL pointers.

NOTE

The maximum length of data that can be sent in EESL_send_event() function is 1024 bytes including the header.

Sending Custom Events

Custom events can be sent between any two applications registered with VMAC. An application is uniquely identified by its logical application name.

VMAC provides a function to send custom events between two applications.

EESL_send_event()

Sends a custom application event and optionally, event data to a destination task already registered with VMAC.

Prototype

```
short EESL_send_event(char *destinationTask,
                      unsigned short customEESLEvent,
                      unsigned char *customEESLEventData,
                      unsigned short dataLength);
```

Parameters

`destinationTask`	Application logical name of the destination task.
`customEESLEvent`	The custom event being sent.
`customEESLEventData`	The associated custom event data. This is set to null if there is no associated event data.
`dataLength`	Size of the custom event data.

Return Values

Returns one of following non-zero values on error:

Value	Description
EESL_SEND_FAILURE	Failed to send the custom event to the target application. All custom events are sent using a send, acknowledge protocol which is transparent to the EESL user. An error of this kind indicates that the target application was unable to acknowledge the message containing the Custom Event. The target application must acknowledge the incoming message within 800ms, or EESL_send_event() function will fail. Possible cause: The target application has not made a call to `wait_event()` or `peek_event()` function within the 800 ms window.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.
EESL_SEND_REJECT	The target application has rejected the custom event because its custom event queue is full (refer to the Custom Event Classes section for more details).

Value	Description
EESL_APP_NOT_FOUND	The destination application specified has not registered.
EESL_APPLICATION_BUSY	The target application is not ready to receive `CLASS - A` events (refer to the Custom Event Classes section for more details).

NOTE

To send an event to the Device Manager to acquire devices, it is recommended to use EESL_send_devman_event() function.

Sending Events to Tasks Using Device Manager

This call in this section deals with sending events through the Device Manager.

EESL_send_devman_event()

Sends a custom application event and optionally, event data to a destination task that has already registered with VMAC via Device Manager. This is required when some devices are required as part of the event. This event must be present in the device mapping table of the receiver.

```
(HIGH, APP_CUST_REQUEST_EVENT, // input event
APP_CUST_EVENT, // output event
{(CONSOLE),(COMM_1),(MAG_READER)}), // devices required
```

The custom application event `APP_CUST_REQUEST_EVENT` is sent to Device Manager, which in turn will send the output event `APP_CUST_EVENT` to the destination task if the devices are available. The sender gets `APP_RES_SET_UNAVAILABLE` if the devices required are not available.

Prototype

```
short EESL_send_devman_event(char *destinationTask,
                             unsigned short customEESLEvent,
                             unsigned char *customEESLEventData,
                             unsigned short dataLength);
```

Parameters

`destinationTask`	Application logical name of the destination task.
`customEESLEvent`	The custom event being sent.
`customEESLEventData`	The associated custom event data. This is set to null if there is no associated event data.
`dataLength`	Size of the custom event data.

Return Values

Returns one of following non-zero values on error:

Value	Description
EESL_SEND_FAILURE	Failed to send the custom event to the target application. All custom events are sent using a send, acknowledge protocol which is transparent to the EESL user. An error of this kind indicates that the target application was unable to acknowledge the message containing the custom event. The target application must acknowledge the incoming message within 800 ms, or EESL_send_event() function will fail. Possible cause: The target application has not made a call to `wait_event()`, or `peek_event()` function within the 800 ms window.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.
EESL_SEND_REJECT	The target application has rejected the custom event because its custom event queue is full (refer to the Custom Event Classes section for more details).
EESL_APP_NOT_FOUND	The destination application specified has not registered.
EESL_APPLICATION_BUSY	The target application is not ready to receive `CLASS – A` events (refer to the Custom Event Classes section for more details).

Sending More than One Event to Device Manager

More than one events can be sent to the Device Manager along with releasing devices using `EESL_send_res_release_event` or `EESL_send_res_release_eventEx`.

EESL_send_res_release_event()

Informs the Device Manager the devices being released in `devReleased` and sends events specified by the variable argument list of type `EESL_EVENT` as event data to the Device Manager.

The events can be either request events for itself events to other tasks that require devices events to the Device Manager (for example, `RES_REJECT_EVENT`, `DISABLE_EVENT_ACTION`, `ENABLE_EVENT_ACTION`, and `DEVICE_NOT_PRESENT_EVENT`). Events that *cannot* be piggybacked are `RES_COMPLETE_EVENT` and <device>`_RELEASED_EVENT`, as the devices being released, which are informed by `devReleased`. These events are ignored by the Device Manager if sent piggybacked.

Prototype

```
long EESL_send_res_release_event(unsigned short devReleased, EESL_EVENT *
                                 eEvent, ...);
```

Parameters

`devReleased`	Devices released.
`eEvent`	Variable arguments of type `EESL_EVENT` to send to the Device Manager.

Return Values

Returns one of following non-zero values if an error occurs:

Value	Description
EESL_SEND_FAILURE	Failed to send the custom event to the target application. All custom events are sent using a send, acknowledge protocol transparent to the EESL user. An error of this kind indicates that the target application was unable to acknowledge the message containing the Custom Event. The target application must acknowledge the incoming message within 800 ms, or EESL_send_event fails. Possible cause: The target application has not made a call to `wait_event()` or `peek_event()` function within the 800 ms window.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.
EESL_SEND_REJECT	The target application has rejected the custom event because its custom event queue is full (refer to the Custom Event Classes section for more details).
EESL_APP_NOT_FOUND	The destination application specified has not registered.
EESL_APPLICATION_BUSY	The target application is not ready to receive `CLASS – A` events (refer to the Custom Event Classes section for more details).
EESL_FAILURE	The total size of the events, their sizes, and the echo data if any which EESL layer appends to an event, exceeds the maximum event data size specified by the application. (`MAX_CUSTOM_EVENT_DATA` or `MAX_CUSTOM_EVENT_DATA_EX`).

EESL_EVENT structure is defined in `eeslapi.h` header file as:

```
typedef struct eesl_event
{
   unsigned short customEESLEvent; // event ID
   unsigned short evtDataLength;        // event data length
   char destinationTask[EESL_APP_LOGICAL_NAME_SIZE];
   //destination task name
   unsigned char * EESLEventData; //event data
}EESL_EVENT;
```

EESL_send_res_release_eventEx()

Informs the Device Manager about the devices being released in `devReleased`, and sends events specified by the variable argument list of type `EESL_EVENT` as event data to the Device Manager.

The events can be either request events for itself or events to other tasks that require devices events to the Device Manager (for example, `RES_REJECT_EVENT`, `DISABLE_EVENT_ACTION`, `ENABLE_EVENT_ACTION` and `DEVICE_NOT_PRESENT_EVENT`). Events that *cannot* be piggybacked are `RES_COMPLETE_EVENT` and <device>`_RELEASED_EVENT`, as the devices being released, which are informed by `devReleased`. These events are ignored by the Device Manager if sent piggybacked.

Prototype

```
long EESL_send_res_release_eventEx(unsigned long devReleased,
                                   EESL_EVENT * eEvent, ...);
```

Parameters

`devReleased`	Devices released.
`eEvent`	Variable arguments of type `EESL_EVENT` to send to the Device Manager.

Return Values

Returns one of following non-zero values if an error occurs:

Value	Description
EESL_SEND_FAILURE	Failed to send the custom event to the target application. All custom events are sent using a send, acknowledge protocol transparent to the EESL user. An error of this kind indicates that the target application was unable to acknowledge the message containing the Custom Event. The target application must acknowledge the incoming message within 800 ms, or EESL_send_event fails. Possible cause: The target application has not made a call to `wait_event()` or `peek_event()` function within the 800 ms window.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.
EESL_SEND_REJECT	The target application has rejected the custom event because its custom event queue is full (refer to the Custom Event Classes section for more details).
EESL_APP_NOT_FOUND	The destination application specified has not registered.

Value	Description
EESL_APPLICATION_BUSY	The target application is not ready to receive `CLASS – A` events (refer to the Custom Event Classes section for more details).
EESL_FAILURE	The total size of the events, their sizes, and the echo data if any which EESL layer appends to an event, exceeds the maximum event data size specified by the application. (`MAX_CUSTOM_EVENT_DATA` or `MAX_CUSTOM_EVENT_DATA_EX`).

EESL_EVENT structure is defined in `eeslapi.h` header file as:

```
typedef struct eesl_event
{
   unsigned short customEESLEvent; // event ID
   unsigned short evtDataLength;        // event data length
   char destinationTask[EESL_APP_LOGICAL_NAME_SIZE];
   //destination task name
   unsigned char * EESLEventData; //event data
}EESL_EVENT;
```

Receiving Custom Events

The EESL library queues incoming custom events. The queue stores up to 30 custom events and associated custom event data depending on the size of the event data for each event. The larger the blocks of event data, the fewer the number of custom events queued. Each new custom event received is added to the end of the custom event queue. If the queue is full, EESL rejects any other incoming custom events until space is available. Application developers access the contents of this queue through an EESL API function EESL_read_cust_evt() function.

Custom events are transferred between applications using the pipe functionality provided by the Verix eVo OS. So, whenever a custom event has arrived at an application, the wait_event(), read_event(), wait_evt(), read_evt() functions return the pipe event (`EVT_PIPE`).

If a pipe event (`EVT_PIPE`) is received, check the custom event queue to see if any custom events are available (refer to the EESL_read_cust_evt() function for more details).

Custom Event Receive/Read Functions

To make the use of the EESL as transparent as possible, wait_event(), read_event(), wait_evt(), read_evt() functions are provided that override the existing OS wait_event(), read_event(), wait_evt(), read_evt() functions. From the caller point of view, the behavior of these functions is unchanged from the functions provided by the OS. However, additional processing occurs in the background.

wait_event(), read_event(), wait_evt(), read_evt()

These functions operate in the same way as the Verix eVo OS functions they replace. These enable `EESL` to check for custom events, and populate the custom event queue. `EESL` does not receive custom events if these functions are not called.

Prototype

```
event_t wait_event();
event_t read_event();
long wait_evt (long interesting_events);
long read_evt (long interesting_events);
```

Return Values

Refer to the respective functions in the *Verix V Operating System Programmers Manual*.

NOTE

If `EVT_PIPE` event is received, check the custom event queue, using the EESL_read_cust_evt() EESL API function.

EESL_read_cust_evt()

Retrieves a custom event and associated event data from the custom event queue.

Prototype

```
unsigned short  EESL_read_cust_evt(unsigned char * eventDataBuffer,short
                                   bufferSize,unsigned short  *
                                   eventDataSize,char * senderLogName);
```

Parameters

`eventDataBuffer`	If this parameter is not null, then the event data is copied into the buffer provided and the event is deleted from the queue. If this parameter is null, then the API returns `EESL_INVALID_PARAMETER` and the event is *not* deleted from queue. If this parameter is not null and the size is not enough, then the API returns `EESL_ERROR_BUFFER_TOO_SMALL`, and the event is deleted from queue.
`bufferSize`	The size of the `eventDataBuffer`. If this is smaller than the size of the event data, the data will not be copied to the caller's buffer.
`eventDataSize`	The return value is the size of the event data.
`senderLogName`	The logical name of the application that sent the event.

Return Values

- Returns the first custom event at the front of the queue. Returns 0, if the queue is empty or if VMAC is not present.
- Returns `EESL_INVALID_PARAMETER` if invalid pointers are passed.
- If the value returned is `EESL_ERROR_BUFFER_TOO_SMALL` then `eventDataSize` is assigned with the actual size of the data in the event. Hence application can increase the size of the buffer and call `EESL_read_cust_evt` again to retrieve the data.

EESL_read_evt_of_type()

Searches for the specified event in the event queue. If the event is found, it returns the event with the associated data. However, the queue is not altered after retrieving the data.

Prototype

```
unsigned short EESL_read_evt_of_type(unsigned short eventType, unsigned
char * eventDataBuffer, short bufferSize, unsigned short * eventDataSize,
char * senderLogName);
```

Parameters

`eventType`	The event to be searched in the event queue. If this value is 0, the API returns an EESL_INVALID_PARAMETER.
`eventDataBuffer`	• If this parameter is not null, then the event data is copied into the buffer provided and the event is deleted from the queue. • If this parameter is null, then the API returns EESL_INVALID_PARAMETER and the event is not deleted from the queue. • If this parameter is not null, and the size specified is not enough, then the API returns EESL_ERROR_BUFFER_TOO_SMALL, and the event is deleted from the queue.
`bufferSize`	The size of the eventDataBuffer. If this is smaller than the size of the event data, the data will not be copied to the caller's buffer.
`eventDataSize`	The return value is the size of the event data.
`senderLogName`	The logical name of the application that sent the event.

Return Values

Returns the custom event from anywhere in the queue. Returns 0, if the queue is empty or if VMAC is not present.

Value	Description
EESL_INVALID_PARAMETER	If invalid pointers are passed.
EESL_ERROR_BUFFER_TOO_SMALL	If the size of the buffer for event data is small.
EESL_EVENT_NOT_FOUND	If the event is not found in the queue.

EESL_get_event()

Retrieves a custom event and associated event data from the custom event queue.

Prototype

```
long EESL_get_event(unsigned short  *customEventId, unsigned char
                    *eventData, unsigned short  *eventDataSize, unsigned
                    char *senderLogicalName);
```

Parameters

`customEventId`	The custom event ID is copied into this location.
`bufferSize`	The size of the `eventDataBuffer`. If this is smaller than the size of the event data, the data will not be copied to the caller's buffer.
`eventData`	If this parameter is not null, then the event data is copied into the buffer provided.
`eventDataSize`	The size of the event data is returned in this parameter.
`senderLogicalName`	The logical name of the application that sent the event.

Return Values

Returns the first custom event at the front of the queue. Returns 0, if the queue is empty or 0, if VMAC is not present.

Value	Description
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this returns the same error.
EESL_FAILURE	The queue is empty.
EESL_INVALID_PARAMETER	Invalid parameter was passed to the function check for null pointers.

NOTE

- If the application initializes using the EESL_Initialise() API, then the `eventData` buffer size should be 300 bytes.
- If the application initializes using the EESL_InitialiseEx() API, then the `eventData` buffer size should be 1024 bytes.

EESL_queue_count()

Obtains the number of events in the custom event queue.

Prototype `short EESL_queue_count(void);`

Return Values The number of events is the custom event queue.

EESL_appl_busy()

Call this function if the application is not ready to receive `CLASS - A` (refer to the Custom Event Classes section for more details) events.

NOTE 

It is very important that the application calls this function when it is not able to process `CLASS - A` events. The EESL layer returns `EESL_APPLICATION_BUSY` to any application trying to send a `CLASS - A` custom event.

Prototype `void EESL_appl_busy(void);`

EESL_appl_ready()

Call this function if the application is ready to receive `CLASS - A` (refer to the Custom Event Classes function for more details) events. The default behavior of the `EESL` is that the application is ready to receive `CLASS - A` events.

Prototype `void EESL_appl_ready(void);`

EESL_is_appl_busy()

This API can be used to know whether the target applications are ready or busy to receive the events.

Prototype `int EESL_is_appl_busy(const char *name);`

Parameters

`name` is the application logical name.

Return Values

Value	Description
EESL_INVALID_PARAMETER	If the application logical name is NULL.

Value	Description
EESL_APPLICATION_BUSY	If the application is busy.
EESL_FAILURE	Invalid application name.

VMAC Queries

VMAC provides a functionality to query which applications are present on the terminal. It may occur that specific functionality of an application running under VMAC is dependent on the other applications running on the terminal. This application can send a query to find out if any application it is dependant upon, is present. For example, an application may not display a menu option “Send mail,” if the mail application required for sending mail is not available on the terminal.

Functions to Query VMAC

This section presents the query functions.

EESL_check_app_present()

Checks if an application has registered with VMAC (refer to the EESL_Initialise() function for more details).

Prototype

```
short EESL_check_app_present(char *logicalName, EESL_IMM_DATA * appData);
```

Parameters

`LogicalName`	Application logical name of the task to which the inquiry is directed.
`appData`	If the application runs and this pointer is not null, then this structure (`EESL_IMM_DATA`) is filled by the function. Refer to the EESL_IMM_DATA section for more details.

Return Values

Value	Description
EESL_APP_NOT_FOUND	VMAC did not run this application.
EESL_APP_STARTED	The specified application is not registered with VMAC. However, the application was run by IMM, and therefore has an entry in the `default.INI`.
EESL_APP_REGISTERED	The application specified has registered with VMAC.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this return the same error.
EESL_INVALID_PARAMETER	An invalid parameter was passed to the function check for null pointers.

EESL_IMM_DATA

The definition of `EESL_IMM_DATA` is as follows:

```
/* Task information structure */
typedef struct EESL_IMM_DATA
{
   char appLogicalName[EESL_APP_LOGICAL_NAME_SIZE];
   task_status_t status;
   unsigned char gid;
   unsigned char logicalId;
   unsigned char programId;
}EESL_IMM_DATA;
```

Field description

Value	Description
`LogicalAppName`	The logical name of the application.
`Gid`	The GID where the application resides.
`LogicalId`	The integer ID assigned by IMM.
`programId`	The integer ID assigned to the application by the Verix eVo OS.

`status`	A set of bits describing the status of the application. There are two elements of task_status_t that are of use to developers:
	`typedef struct task_status_t`
	`{`
	`unsigned running :1`
	`unsigned registered :1`
	`…..`
	`} task_status_t;`

The EESL return values are defined as follows:

#define	Value
EESL_ERROR_INIT_FAILURE	10
EESL_ERROR_REG_FAILURE	11
EESL_SEND_FAILURE	12
EESL_SEND_REJECT	13
EESL_PIPE_READ_ERROR	14
EESL_INVALID_PARAMETER	15
EESL_APP_NOT_FOUND	16
EESL_APP_REGISTERED	17
EESL_APP_STARTED	18
EESL_ERROR_BUFFER_TOO_SMALL	19
EESL_FAILURE	20

Advanced EESL Usage

Additional EESL Functions

The following is the list of APIs provided by VMAC.

EESL_get_outgoing_flexi()

Retrieves the outgoing flexi-record. Developer specific fields can be added to the flexi-record before it is sent using EESL_send_event() function.

CAUTION

Do not use reserved VMAC message Field IDs (Refer to the Special EESL Flexi-Record IDs section for more details).

All messages sent between applications in the VMAC environment use the Flexi-record format (refer to the Flexi-Records chapter for more details). The custom application event and the associated custom event data are stored in individual flexi-record fields. The outgoing flexi-record is stored in `EESL` global space. The flexi-record is reset (all fields deleted) after a call to EESL_send_event() function, and after a call to EESL_Initialise() function.

Prototype

```
Unsigned char * EESL_get_outgoing_flexi();
```

EESL_get_incoming_flexi()

Retrieves the entire custom event flexi-record from the beginning of the EESL custom event queue.

Prototype

```
short EESL_get_incoming_flexi(unsigned char * flexiBuffer, short
                              bufferSize,char * senderLogName);
```

Parameters

`flexiBuffer`	The message flexi-record, at the front of the queue, is copied to this buffer. The queue is decreased by one.
`BufferSize`	The size of the caller's buffer. If the buffer is too small, an error will occur.
`SenderLogName`	The logical name of the application that sent the message.

Return Values

The function returns 0, if no error occurs. It returns the following non-zero value if an error occurs.

Value	Description
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.
EESL_FAILURE	The queue is empty.
EESL_ERROR_BUFFER_TOO_SMALL	The buffer provided is too small to copy in the message flexi-record.
EESL_INVALID_PARAMETER	Invalid parameter was passed to the function check for NULL pointers.

EESL_logical_name_list()

Retrieves the list of logical application names of applications started by VMAC.

Prototype

```
short EESL_logical_name_list(short  * numberTasks, unsigned char *list,
                             short  listBufferSize, short  *
                             receivedBufferSize);
```

Parameters

`numberTasks`	The number of tasks active under VMAC is returned in this parameter.
`list`	An array of structures, of type `EESL_IMM_DATA`, are copied into this caller allocated buffer.
`listBufferSize`	The size of the buffer provided by the caller. The list of Logical Application names are truncated if the buffer provided is too small.
`receivedBuffSize`	The total size of the array of structures (of type `EESL_IMM_DATA`) received, in bytes.

NOTE

If the `listBufferSize` is smaller than the list of tasks, the buffer contains only the tasks it can accommodate, but `numberTasks` contains the total number of active tasks.

Return Values

Returns the one of following non-zero values if an error occurs:

Value	Description
EESL_FAILURE	Communication with VMAC failed. For reasons unknown, a response was not forthcoming from the IMM.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this return the same error.

EESL_get_task_list()

Retrieves the list of logical application names of applications successfully run by VMAC.

Prototype

```
short EESL_get_task_list(short  * numberTasks, unsigned char *list, short
                         listBufferSize, short  * receivedBufferSize);
```

Parameters

`numberTasks`	The number of tasks active under VMAC is returned in this parameter.
`list`	A pointer to an array of structures (of type EESL_IMM_DATA) is copied into this caller allocated buffer.
`ListBufferSize`	The size of the buffer provided by the caller. The list of Logical Application names are truncated if the buffer provided is too small.
`ReceivedBuffSize`	The total size of the array of structures (of type `EESL_IMM_DATA`) received, in bytes.

Return Values

Returns the one of following non-zero values on error:

Value	Description
EESL_FAILURE	Communication with VMAC failed. For reasons unknown, a response was not forthcoming from the IMM.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this will return the same error.

NOTE

The list should an array of type `EESL_IMM_DATA`, type cast to `unsigned char *`.

EESL_get_number_of_tasks()

Retrieves the number of applications that are successfully run by VMAC.

Prototype

```
short EESL_get_number_of_tasks(short  * numberTasks);
```

Return Values

Returns the one of following non-zero values on error:

Value	Description
EESL_FAILURE	Communication with VMAC failed. For reasons unknown, a response was not forthcoming from the IMM.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this return the same error.

EESL_set_send_retries()

Sets the number of attempts `EESL_send_event` makes to receive an acknowledgement from the receiving task. For each attempt, EESL_send_event() function sends the event, and waits 200 ms for a response from the receiving application.

Prototype

```
void EESL_set_send_retries(short  sendRetries);
```

Parameter

`SendRetries` The number of send retries to attempt. The default is 3.

Special EESL Flexi-Record IDs

Knowledge of this section is not required by application developers.

Custom event messages use the Flexi-record format (refer to the EESL_get_outgoing_flexi() function and EESL_get_incoming_flexi() function).

The following flexi-record IDs have already been used:

```
/* EESL Flexi Record Message ID's */
#define EESL_ID_EVENT    1
#define EESL_ID_EVENT_DATA 2
#define EESL_MESSAGE_COUNT_ID 3
#define EESL_REPEAT_INDICATOR 4
#define EESL_LINK_COUNTER 5
#define CUSTOM_EVENT_ID_OS_EVENT  6
#define EESL_STORE_RETURN_DATA 8
#define EESL_ECHO_DATA   9
#define EESL_SECONDARY_TASK_ID 10
```

An EESL flexi-record ID called `EESL_STORE_RETURN_DATA` has a special use for application developers. The data in this flexi-record field is echoed back to the calling application whenever the target application sends an event back to the calling application. The data is returned in a flexi field with ID `EESL_ECHO_DATA`. The field is not added by the EESL, and must be added by application developers through EESL_get_outgoing_flexi() function.

When the EESL layer, in application B, receives a flexi-record from application A, with the `EESL_STORE_RETURN_DATA` field ID set, the associated field data is read and stored by the EESL layer. The fact that specific field data is stored is transparent to the application.

When the application B sends any custom event back to application A, the stored data added to the outgoing message in a field with ID `EESL_ECHO_DATA`.

The echo data stored has a maximum size of 30 bytes.

NOTE

A second consecutive message containing the `EESL_STORE_RETURN_DATA` field overwrites any field data stored from a previous message.

Device Manager

The Device Manager is an integral part of the Multi Application architecture and is dependent on the IMM and the EESL for its operation.

The Device Manager is responsible for device (resource) management within the Verix eVo terminals. Device management, as defined in this document, refers to the controlled access of all the hardware devices available in the Verix eVo range of terminals. These devices are accessed through the OS interface API (through the Verix V SDK), or library layer APIs above the OS.

These devices include:

- Console
- Contactless Card Reader
- Modem
- Communication ports
- Magnetic stripe reader
- Smart card readers
- Printer
- Ethernet
- USB Serial (USBSER_1)
- USB UART (DEV_COM6)

Device management does not include:

- file access control
- pipe access control
- direct control of any devices external to the terminal, such as external PIN pads.

The Device Manager is the first task started by the IMM. This means it must be the first task specified in the `default.INI` file. The Device Manager opens all devices before registering with the IMM. This forces all applications in the Multi-Application environment to cooperate with the Device Manager in order to acquire devices.

It controls device access in applications by sending custom events to each application, through the EESL layer. It has a very open architecture in the sense that each application running in the terminal is responsible for providing a device mapping table to the Device Manager. The device mapping table determines device allocation for each application. The Device Manager is a single Verix eVo executable DEVMAN.OUT, and runs in GID1 on Verix eVo terminals, along with all other core VMAC executables.

Device Management Activities

Device management is based on events. Device management activities consist of device availability checks and sending events. The Device Manager receives request events and responds with events.

Availability of the devices requested is checked and the response event is sent to the destination task. The Device Manager takes an all or nothing approach for device allocation activities, all the devices specified, must be available before the Device Manager sends the response event. If any one of the devices is not available the task gets an event informing that the devices were not available for the request event sent.

NOTE

There is an exception to the device allocation activity. Refer to the Handling Devices Not Present on the Terminal section for more details about the exception.

Device Mapping Table

Each task in the VMAC environment must provide a device mapping table to request for any devices it needs from Device Manager.

The application needs to define all the events for:

- Requesting a device
- Changing the device owner
- Activating a task
- Sending an event to another task along with allocation of devices.

The device mapping table is generated using the Resource Compiler tool (rck2.exe). The name of the device mapping resource File is <Application Logical name>.RES

or,

The device mapping resource file name could be specified by the CONFIG.SYS variable of the format

```
#<Application Logical Name>=<Device mapping resource file name>
```

- <Application Logical Name> is the logical application name specified in the default.INI. The file extension is always .RES.
- <Device mapping resource file name> is the base name of the device mapping resource with length <=8 characters excluding “f:” or “\” or “f:\” specifying the path of the resource file.

For example, if the logical name is `APPNAME`, then the environment variable is `#APPNAME`. The value of the environment variable must be the base name of the device mapping resource without the extension. For `APPNAME`, if the resource file name is `APPres.res`, the environment variable is `#APPNAME=APPres`. It could also give the path of the file as used with the RM_Open call of the resource manager. If the file is downloaded to the flash, the environment variable could be specified as `#APPNAME=f:APPres`

The Device Manager queries the `EESL` for the logical names of all applications. The Device Manager then searches for each device mapping file in the GID in which the application executable resides.

The device mapping table consists of one or more records. Each record specifies a device management functional unit that may consist of one or more device management activities to be carried out by the Device Manager. These device management activities are all carried out on the application which owns the device mapping table. The application that owns a specific device mapping table is called the *target* application.

Device management activities are divided into two categories:

- Device check: Determines if the devices that the application requires for an input event are available.
- Sending Events: Sends custom application events to the target application.

Devices and output events are both represented by integer values in the device mapping table.

NOTE

The Device Manager makes no provision for queuing devices required by more than one application.

The following defines the elements of a single record in the device mapping table:

- Event priority
- Input event
- Output event
- List of devices/actions

Event Priority

Events could be assigned high, medium, or low priority. Because each executable has a device mapping table, a single application could contain functionality that falls into one or more of these categories. If there is device contention between an event of higher priority and an event of lower priority, the device is assigned to the event with the higher priority. For example, if device contention exists between a medium priority event and a low priority event, the device is awarded to the medium priority event.

When a device requested by a high priority event is owned by a low priority event, a request (custom event) is sent to the application to release the device. It is up to the application to release the device.

Table 3 Priority Table

Priority Type	# define	Value	Description
Mission Critical Status	High	1	Mission critical events include all financial transaction activities such as authorization and settlement of financial applications. These events have highest priority and are rewarded devices over any other events with lower priorities.
Diagnostic Status	Medium	2	This category includes terminal management and diagnostic activities. The Device Manager awards resources to diagnostic status events over devices over value added status activities and the Device Manager requests events with diagnostic status to relinquish all devices if there is a requirement from a mission critical status event.
Value Added Status	Low	3	This category includes all events that perform noncritical functionality such as email and calculator. The Device Manager requests events with value added status to relinquish all devices if there is a requirement from a diagnostic status or mission critical status event.

NOTE

The priority values are part of the header file `<EVOVMAC>\include\devman.h`.

Input Events

The Device Manager receives custom application events from other applications. The device map input events are custom application events. Each input event acts as a key to a specific record in the table.

An application has to send an event to the Device Manager to enable it to carry out a device availability check and generate an output event. This event sent by the application is called input event.

Output Events

The Device Manager sends the output events to the applications after performing the device management activities.

Output events are application-defined custom events that are handled by the application. This event indicates that the devices requested by the application are available to it.

List of Devices/ Actions

Each record in the device mapping table specifies a list of devices that the application requires to claim. In addition to the devices the list could also have the change owner action that indicates the transfer of open devices to the application or activate action that indicates that the task should be activated by the `activate_task()` OS call, if the console is available.

Template for Device Mapping Table

The device mapping table is written as a resource with resource ID `DEVMAN_RSRC` in <Application logical name>`.rck` file as per the template specified by the Device Manager (`<EVOVMAC>\template\DevMan.rt`) and compiled using `rck2.exe` utility. The template file is as follows:

```
template DevManRsrc TEMPLATE_DEVMAN_RSRC
{
    // internally updated by resource manager
    short numOfInEvt = $countof(evtDeviceList);
    array evtDeviceList
    {
        short priority,// priority: HIGH, MEDIUM, LOW
        unsigned short inputEvent; // input event
        unsigned short outputEvent; // output event
        // internally updated by resource manager
        short numOfAct = $countof(reqActions);
        array reqActions // list of required devices,
        {
        short action; // device /change owner actions/Activate action
        }
    }
};
```

The application must define events. This can be when it requires a device or change owner action or activate action and the output event to be sent to the application by the Device Manager in case all these devices are available.

Sample Device Mapping Table

```
// DEVMAN_RSRC resource

resource app1Rsrc DEVMAN_RSRC DevManRsrc
{
    {
    // event 1
    (HIGH, // priority of the event
    APPLICATION_INIT_EVENT, // input event
    INIT_EVENT, // output event
        {(CONSOLE),(COMM_1),(MAG_READER)}), // devices required

    // event 2
    (HIGH, //priority of the event
    APPLICATION_ACTIVATE_EVENT, // input event
    APPLICATION_ACTIVATE_EVENT, // output event
```

```
    {(ACTIVATE_ACTION)}), // devices required
    // event 3
    (HIGH, / /priority of the event
    COMMUNICATION_EVENT, // input event
    COMM_3_TRANSFER_EVENT, // output event
    {(COMM_3)}), // devices required
    //event 4
    (MEDIUM, //priority of the event
    PRINTER_EVENT, // input event
    PRINTER_EVENT, // output event
    {(COMM_4)}) // devices required
    }
};
```

NOTE

The resource ID is `DEVMAN_RSRC`, which is defined by the Device Manager. The # defines for the devices, the App response events and the `ACTIVATE_ACTION` are available in `VMAC.h` header file. The other # defines and events specified by the Device Manager are available in the `<EVOVMAC>\include\devman.h` header file.

Compile the resource file with the following command:

```
<EVOVMAC>\tools\rck2 -sAppName -oAppName -M
```

where, `AppName` is the application logical name.

The `.res` file should be downloaded in the same GID as the application.

Device Values

The following are the values for devices that are used by the Device Manager. These values are available as part of the `VMAC.h` or `devman.h` header file.

Device	Value
CONSOLE	0
MAG_READER	1
BEEPER	2
CLOCK	3
BAR_CODE	4
COMM_1	5
COMM_2	6
COMM_3	7
COMM_4	8
COMM_5	9
ICC_1	10
ICC_2	11
ETH1	12
NO_DEVICES_REQUIRED	13
WLAN1	14
COMM_6	15

Device	Value
USBD_1	16
CRYPTO	17
CTLS1	18
COMM_8	19
USBSER_1	20

A configuration variable `CRYPTO` should be downloaded to GID1 and set to 1 to use the functionality of CRYPTO device.

If an application has an event that does not require any device, the device map entry is as follows:

```
(HIGH, APP_I_EVENT, APP_O_EVENT, {(NO_DEVICES_REQUIRED)})
```

If the device list contains NO_DEVICES_REQUIRED, the whole record will be ignored. For example,

```
(HIGH, APP_I_EVENT, APP_O_EVENT,
{(NO_DEVICES_REQUIRED),(COMM_3),(COMM_4)})
```

Whenever the application acquires WLAN1, the application should verify whether the device is enumerated or not by using `get_usb_device_bits()` OS API.

The application should call the OS API `set_event_bit()` after opening `CTLS1` to enable the device to receive events.

Device Mask Values

Device mask values are bit mask values for the devices. These values indicate the devices in the bitmap value provided in response to `DEVICE_NOT_PRESENT_DATA` and indicate the devices being released in `EESL_send_res_release_event`.

Device Mask	Value
CONSOLE_MASK	0x0001
MAG_READER_MASK	0x0002
BEEPER_MASK	0x0004
CLOCK_MASK	0x0008
BAR_CODE_MASK	0x0010
COMM_1_MASK	0x0020
COMM_2_MASK	0x0040
COMM_3_MASK	0x0080
COMM_4_MASK	0x0100
COMM_5_MASK	0x0200
ICC_1_MASK	0x0400
ICC_2_MASK	0x0800
ETH1_MASK	0x1000
WLAN1_MASK	0x4000
COMM_6_MASK	0x8000
USBD_1_MASK	0x10000

Device Mask	Value
CRYPTO_MASK	0x20000
CTLS_1_MASK	0x40000
COMM_8_MASK	0x80000
USBSER_1_MASK	0X100000

Device Manager Generated Events

The Device Manager generates certain events. The Device Manager sends these events to an application without any request event from the application. The Device Manager events are in the range 20001-20500. These are `CLASS - A` type events of EESL, i.e., they must be processed by the application immediately.

The Device Manager generates an `APPLICATION_INIT_EVENT` at system startup.

This event gives all applications a chance to allocate the devices they need for initialization. This is only sent if the task has listed this in its device mapping table and the devices requested are available.

The device available events are sent when a device is available and is not requested by any other application. The available event must be listed in the device mapping table of that application.

The device request event is sent when a device is with an event that has a priority lower than that of the event requesting the device. The request events will not be listed in the device mapping table and is only sent by Device Manager. `APPLICATION_INIT_EVENT` is the event generated by the Device Manager:

Device Available Events

The device available events are:

- `CONSOLE_AVAILABLE_EVENT`
- `MAG_READER_AVAILABLE_EVENT`
- `BEEPER_AVAILABLE_EVENT`
- `CLOCK_AVAILABLE_EVENT`
- `BAR_CODE_AVAILABLE_EVENT`
- `COMM_1_AVAILABLE_EVENT`
- `COMM_2_AVAILABLE_EVENT`
- `COMM_3_AVAILABLE_EVENT`
- `COMM_4_AVAILABLE_EVENT`
- `COMM_5_AVAILABLE_EVENT`
- `ICC_1_AVAILABLE_EVENT`
- `ICC_2_AVAILABLE_EVENT`
- `ETH_1_AVAILABLE_EVENT`
- `WLAN_1_AVAILABLE_EVENT`

- COMM_6_AVAILABLE_EVENT
- USBD_1_AVAILABLE_EVENT
- CRYPTO_AVAILABLE_EVENT
- CTLS_1_AVAILABLE_EVENT
- COMM_8_AVAILABLE_EVENT
- USBSER_1_AVAILABLE_EVENT

Device Request Events

The device request events are:

- CONSOLE_REQUEST_EVENT
- MAG_READER_REQUEST_EVENT
- BEEPER_REQUEST_EVENT
- CLOCK_REQUEST_EVENT
- BAR_CODE_REQUEST_EVENT
- COMM_1_REQUEST_EVENT
- COMM_2_REQUEST_EVENT
- COMM_3_REQUEST_EVENT
- COMM_4_REQUEST_EVENT
- COMM_5_REQUEST_EVENT
- ICC_1_REQUEST_EVENT
- ICC_2_REQUEST_EVENT
- ETH_1_REQUEST_EVENT
- WLAN_1_REQUEST_EVENT
- COMM_6_REQUEST_EVENT
- USBD_1_REQUEST_EVENT
- CTLS_1_REQUEST_EVENT
- CRYPTO_REQUEST_EVENT
- COMM_8_REQUEST_EVENT
- USBSER_1_REQUEST_EVENT

Table 4 Device Request Events Config Variable

Config Variable	Definition
#EVTWAITTIME	Time out, in millisecond, for which device manager will wait for responses from application to release a device before sending APP_RES_SET_UNAVAILABLE to requestor. If this config variable is not present, then 3 seconds is taken as the default value.

Configuration Options

The total size of the mapping table maintained by the Device Manager can be adjusted using the configuration variable `*VMACFLEXRECSIZE`. This variable can be set with a value from 3000 to 5000. The default value is 3000 bytes.

The maximum resource file size for any VMAC application can also be adjusted using the configuration variable `*VMACRESSIZE`. This variable can be set with a value from 200 bytes to 500 bytes. The default value is 350 bytes.

If an application wants to set its own resource file size and not use the one specified by `*VMACRESSIZE`, set the configuration variable `APPNAME_RESSIZE` in the same GID as the application. The `APPNAME` prefix refers to the VMAC logical name of the application. Example, for the VMAC application named `EASYID`, the configuration variable name expected is `EASYID_RESSIZE`. The valid set of values for the configuration variable is the same with `*VMACRESSIZE`.

NOTE

`*VMACFLEXRECSIZE` and `*VMACRESSIZE` configuration variables should be set in GID1.

Acknowledgement Events

Acknowledgment events are a reserved set of custom application events. The application sends acknowledgment events to the Device Manager when it has released all the devices that have been allocated to it, or to reject the devices allocated. The Device Manager sends one or more acknowledgment events to the application that sent an input event. This is to inform the sender application the result of the processing of the device management activities.

The following are the acknowledgment events:

RES_COMPLETE_EVENT

The application sends this event to the Device Manager to release all the devices that the Device Manager had allocated to the application. In the event data of the `RES_COMPLETE_EVENT`, the application should send the event ID of the corresponding output event in the device map of the RCK file.

NOTE

While sending a RES_COMPLETE_EVENT and RES_REJECT_EVENT, the third parameter of EESL_send_event() function should be of 2 bytes, which is the size of short.

Example

```
unsigned short oEvent = PRINT_EVENT;
EESL_send_event("DEVMAN", RES_COMPLETE_EVENT,
                (unsigned char*)&oEvent,sizeof(short));
```

Number of Applications DEVMAN can support

The number of applications DEVMAN can support is dependent on the number of entries in an RCK file (refer to the Device Mapping Table section for more details).

The total size of the Master Device mapping table DEVMAN maintains (M) = 3000 bytes. As the Master Device mapping table is a flexi-record, the header occupies 10 bytes. As a result, the total amount of space available is 3000-10 = 2990 bytes.

Maximum size of <<application's logical name>>.RES file (AR) = 350 bytes.

NOTE

- The total size of the Master Device Mapping Table of DEVMAN can be changed using the *VMACFLEXRECSIZE configuration variable.
- The maximum resource file size can be changed using the configuration variable *VMACRESSIZE or the application-specific configuration variable APPNAME_RESSIZE (refer to the Device Mapping Table in the Configuration Options section).

Number of bytes of memory occupied by an application in the Master Device mapping table is 2 * (5 * R + D)+(6 * R) bytes where:

- R = Number of rows in the RCK file.
- D = Sum of the number of device entries in each row in the RCK file and the NO_DEVICES_REQUIRED as one of the devices.
- D = d1 + d2 + … + di + … + dn where:
  - d1 = number of devices in the first row of RCK file.
  - d2 = number of devices in the second row of RCK file.
  - di = number of devices in the i^{th} row of the RCK file.
  - dn = number of devices in the n^{th} row of RCK file.

For example, consider the RCK file of the VMAC FrontEnd.

```
// DEVMAN_RSRC resource
resource app1Rsrc DEVMAN_RSRC DevManRsrc
{
 {
  (LOW,CONSOLE_AVAILABLE_EVENT, CONSOLE_OPEN_EVENT, {(CONSOLE)}),
  (HIGH,PRINTER_REQUEST_EVENT,PRINTER_REQUEST_EVENT,{(COMM_4)})
}
```

In the above example,

R = 2; D = D1+D2; D1 = 1; D2 = 1; therefore, D = 2

2 * (5 * R + D) + (6 * R) = 2 * (5 * 2 + 2) + (6 * 2) = 36 bytes

Therefore, the size of the Master Device mapping table occupied by VMAC is 46 bytes (36 bytes by FrontEnd + 10 bytes of flexi-record header).

If VMAC is the only application downloaded to the terminal, it occupies 46 bytes of the memory allocated for the Master Device mapping table. The remaining space available for other applications in the Master Device mapping table is:

3000 bytes – 46 bytes = 2954 bytes.

NOTE

As soon as the application receives EVT_SHUTDOWN event, the terminal will shut down. As a result, the VMAC based application need not send any resource release events to DEVMAN.

RES_REJECT_EVENT

The application sends this event to the Device Manager to reject an output event if it was not requested by the application. Similar to `RES_COMPLETE_EVENT`, the application should send the `eventId` for the `RES_REJECT_EVENT` in the event data. This is the event which it received. The application could reject `<device>_AVAILABLE_EVENT` if the task does not want this device, `<device>_REQUEST_EVENT` if the task does not want to release a device requested by Device Manager.

Example

```
unsigned short oEvent = PRINT_EVENT;
EESL_send_event("DEVMAN", RES_REJECT_EVENT,
                (unsigned char*)&oEvent,sizeof(short));
```

EVENT_NOT_HANDLED

The Device Manager sends this event to inform the application that the specified event was not handled. The following are reasons for not handling the event:

- No entry of the event in the Device mapping table for the application. The event Data will contain the `eventId` of the event which was not handled. This is the event which was sent, i.e., the input event in the device map. The next 2 bytes contain the `EVENT_NOT_FOUND` error value.
- Sending event to the application failed. The first 2 bytes of event data will contain the `eventId` of the event which was not handled. This is the event which was sent, i.e., the input event in the device map. The next 2 bytes contain the `EESL_SEND_FAILURE` error value.
- The event has been disabled by the application. The first 2 bytes of event data contains the `eventId` of the event that was not handled. This is the event that was sent, i.e., the input event in the device map. The next 2 bytes contain the `EVENT_DISABLED` error value.

APP_RES_SET_UNAVAILABLE

The Device Manager sends this event to inform the application that the device(s) requested through the output activities event defined in the device mapping table for the application is not available. The event Data will contain the `eventId` of the event which was sent, i.e., the input event in the device map. If the data is more than 2 bytes, from third byte onwards, data is a flexi-record. This flexi-record contains two fields defined in `devman.h` header file.

- **DM_FLD_INPUT_EVENT** - Event that causes this device(s) not available (duplicate of what is there in the first 2 bytes).
- **DM_FLD_RES_NOT_AVL_REASON** - Reason code (short) that specifies the reason for non availability of device. Following are the valid reason codes:

DM_DEVICE_NOT_AVAILABLE	Requested device is not available with VMAC.
DM_SINGLE_USB_NOT_AVAILABLE	Requested device is with VMAC but another application owns another device in single USB mode.

DM_DEVICE_NOT_OWNED	The application does not own any device part of security architecture on which the application can step up security.

Released Events

The device released event is sent by the application to the Device Manager when a device is released by the application either as a response to a device request by the Device Manager or as a partial release of devices.

The following are the released events supported by the Device Manager:

- CONSOLE_RELEASED_EVENT
- MAG_READER_RELEASED_EVENT
- BEEPER_RELEASED_EVENT
- CLOCK_RELEASED_EVENT
- BAR_CODE_RELEASED_EVENT
- COMM_1_RELEASED_EVENT
- COMM_2_RELEASED_EVENT
- COMM_3_RELEASED_EVENT
- COMM_4_RELEASED_EVENT
- COMM_5_RELEASED_EVENT
- ICC_1_RELEASED_EVENT
- ICC_2_RELEASED_EVENT
- ETH_1_RELEASED_EVENT
- WLAN_1_RELEASED_EVENT
- COMM_6_RELEASED_EVENT
- USBD_1_RELEASED_EVENT
- CTLS_1_RELEASED_EVENT
- CRYPTO_RELEASED_EVENT
- COMM_8_RELEASED_EVENT
- USBSER_1_RELEASED_EVENT

Actions

`transfer_action` and `activate_action` are the actions supported by the Device Manager. `transfer_action` transfers the ownership of the devices to the application which asks for the device. After the transfer of the event, the application should delay the opening of the device. The following is a list of `transfer_action` supported by the Device Manager:

- CONSOLE_TRANSFER_ACTION
- MAG_READER_TRANSFER_ACTION
- BEEPER_TRANSFER_ACTION

- CLOCK_TRANSFER_ACTION
- BAR_CODE_TRANSFER_ACTION
- COMM_1_TRANSFER_ACTION
- COMM_2_TRANSFER_ACTION
- COMM_3_TRANSFER_ACTION
- COMM_4_TRANSFER_ACTION
- COMM_5_TRANSFER_ACTION
- ETH_1_TRANSFER_ACTION
- WLAN_1_TRANSFER_ACTION
- COMM_6_TRANSFER_ACTION
- USBD_1_TRANSFER_ACTION
- CTLS_1_TRANSFER_ACTION
- CRYPTO_TRANSFER_ACTION
- COMM_8_TRANSFER_ACTION
- USBSER_1_TRANSFER_ACTION

The following transfer actions are not available when smart card devices are not available:

- ACTIVATE_ACTION

Activate_action activates the application by sending an OS activate event which transfers the console ownership.

Broadcast Events

The Device Manager supports both normal and broadcast input events. Normal input events are input events for each application that have the event defined in the device mapping table, until an application handles the output event. Once an event has been handled, no further device mapping table is looked up. However, on a broadcast input event, all applications that have registered for this input event receive the corresponding output event, irrespective of if other applications handle it.

The normal custom event range for the application is 10001–20000 and the broadcast event range is 30000–31000.

The normal events are `CLASS - A` type events of EESL, i.e., they must be processed by the application immediately. The broadcast events are `CLASS - B` events and can be processed at any time.

Input Event Processing Queue

Assume that more than one application registers for a specific input event, through their respective device mapping tables. Sending this Input Event to the Device Manager causes it to process the event for each application, according to the event priority. To accomplish this, the Device Manager queries all records, in all device mapping tables, that have registered for the current input event and gets the record with highest priority. The Device Manager places this record in the input event processing queue according to the event priority.

The Device Manager processes the input event for each device mapping record in the input event processing queue, according to individual requirements of devices. The target application must reject or send a completion event. When encountered with a reject event, the Device Manager moves on to the next record for the input event.

Sending Events from Tasks

The possible ways of sending events from a task are:

- A task can send an event requesting for devices. In this case, the Device Manager receives the input event listed in the device mapping table of the task. If all devices are available, the task gets the output event. Once the task completes using the devices and no longer requires use of the devices, it must send `RES_COMPLETE_EVENT` with the event received from the Device Manager as the event data.

  The following is an example entry in the device mapping table:

  ```
  ...
      (HIGH, APP_CUST_REQUEST_EVENT, APP_CUST_EVENT,
      {(CONSOLE), (COMM_2)},
  ...
  ```

  When the task sends the event APP_CUST_REQUEST_EVENT

  ```
  retVal = EESL_send_event("DEVMAN",APP_CUST_REQUEST_EVENT, NULL, 0);
  ```

  The `APP_CUST_EVENT`, is sent by the Device Manager if all devices listed for `APP_CUST_REQUEST_EVENT`, console and COMM_2, are available.

  `APP_LOGICAL_NAME` is the application logical name of the task. The task should handle `APP_CUST_EVENT` and send `RES_COMPLETE_EVENT` after closing all the devices as follows:

  ```
  unsigned short eventId;
  eventId = APP_CUST_EVENT;
  retVal = EESL_send_event("DEVMAN",
  RES_COMPLETE_EVENT, (unsigned char*)&eventId, sizeof(short));
  ```

- A task can send an event to another task through the Device Manager. In this case, the Device Manager receives the input event listed in the device mapping table of the task that receives the event. While sending the event the sender has to set the TASK_ID of the receiver in the outgoing flexi-record or use `send_devman_event`. If all the devices are available, the receiver task gets the output event. Once the task completes using the devices and no

longer needs them, it *must* send RES_COMPLETE_EVENT with the event received from the Device Manager as the event data.

For example, an entry in the device mapping table of the task that has to receive the event:

```
...
    (HIGH, APP_CUST_REQUEST_EVENT, APP_CUST_EVENT, {(CONSOLE),
    (COMM_2)},
...
```

When the task sends the event APP_CUST_REQUEST_EVENT

```
retVal = EESL_send_devman_event("APP_LOGICAL_NAME",
                                APP_CUST_REQUEST_EVENT, NULL, 0);
```

where, APP_LOGICAL_NAME is the application logical name of the receiving task.

The APP_CUST_EVENT, is sent to the receiver “APP_LOGICAL_NAME” by the Device Manager if all the devices listed for APP_CUST_REQUEST_EVENT, console and COMM_2, are available.

The task that receives the output event should handle APP_CUST_EVENT and send RES_COMPLETE_EVENT after closing all the devices as follows:

```
unsigned short eventId;
eventId = APP_CUST_EVENT;
retVal = EESL_send_event("DEVMAN",RES_COMPLETE_EVENT,
                         (unsigned char*)&eventId, sizeof(short));
```

If the task does not require this event and rejects the event, the event is sent to the next entry in the table.

- A task can send a broadcast event to all the tasks. In this case, the Device Manager receives the input event listed in the device mapping table of more than one task and all tasks receive the event. This event must have the ID within the range 30000 and 31000. If all devices are available, the receiver task gets the output event. Once the task completes using the devices and no longer needs them, it *must* send RES_COMPLETE_EVENT with the event received from the Device Manager as the event data.

  The following are entries in the device mapping table of the tasks that will receive the event:

```
...
(HIGH, APP_BROADCAST_EVENT, APP_BR_EVENT1,{(CONSOLE),(COMM_2)},
...
(HIGH, APP_BROADCAST_EVENT, APP_BR_EVENT2,{(CONSOLE),(COMM_2)},
...
(HIGH, APP_BROADCAST_EVENT, APP_BR_EVENT3,{(CONSOLE),(COMM_2)},
```

  When an application sends the following event:

```
retVal = EESL_send_event("DEVMAN",APP_BROADCAST_EVENT, NULL, 0);
```

the Device Manager sends the event APP_BR_EVENT1 to all the tasks that have this event listed in their device mapping table.

All tasks that receive the output event should handle the output event (for example, APP_BR_EVENT1, and so on) and send RES_COMPLETE_EVENT after closing all devices:

```
unsigned short eventId;
eventId = APP_BR_EVENT1; // APP_BR_EVENT2, APP_BR_EVENT3
retVal = EESL_send_event("DEVMAN", RES_COMPLETE_EVENT,
                         (unsigned char*)&eventId, sizeof(short));
```

If one or more devices requested are not available, the event remains in the Device Manager event queue and when the device becomes available, the event is sent.

Transfer Open Devices to Other Tasks

Use the following procedure to transfer a device from one task to another without closing the handle.

- Set the owner of the device to the Device Manager, as follows:

```
short retVal;
EESL_IMM_DATA appData;
retVal = EESL_check_app_present("DEVMAN", &appData)
set_owner(handle, appData.programId);
// device handle - return value of device open
EESL_send_event("DEVMAN", APP_COMM_3_EVENT, evtData, evtDataSize);
```

The task that receives the handle has the following entry in its device mapping table:

```
(HIGH, APP_COMM_3_EVENT, APP_COMM_3_EVENT,{(COMM_3_TRANSFER_ACTION)})
```

To get the handle of the device, use either the open() or get_owner() function calls.

```
handle = open("/dev/device", 0);
```

or

```
handle = get_owner(("/dev/<device>", &taskId);
```

where, <`device`> could be any device on the terminal, except smart card devices. `get_owner()` returns the handle and the task ID of the device owner in `taskId`.

NOTE

- Smart card devices (for example, icc1) cannot be transferred from one task to another.
- It is recommended to use `active_task()` function instead of `set_owner()` function to transfer the console ownership.
- Transferring a device which is not owned by the application will result in DM_ILLEGAL_TRANSFER_ACTION error.

Enabling and Disabling Events in the Device Mapping Table

An application that wishes to disable an event sends DISABLE_EVENT_ACTION to the Device Manager. The event data would specify the input event that needs to be disabled. An application that needs to enable an event sends ENABLE_EVENT_ACTION to the Device Manager. The event data would specify the input event that needs to be enabled.

The DISABLE_EVENT_ACTION and ENABLE_EVENT_ACTION have values in the DEVMAN event range. The Device Manager maintains a local copy of the device mapping tables in memory. This is updated with a flag to show if an entry is enabled or disabled.

The original device mapping table (resource file) is untouched, i.e., these changes (enabling/disabling) are not persistent across power cycles, but rather a runtime feature.

NOTE

An application can enable and disable events only for itself.

Handling Devices Not Present on the Terminal

The Device Manager handles the situation of events requesting malfunctioning devices or devices not available on the terminal, in the following way.

The Device Manager sets the status of the device which could not be opened at startup to NOT_PRESENT. The NOT_PRESENT state of the device does not distinguish between the malfunctioning of the device and non-availability of the device on the terminal.

The logport and debugport devices are the exceptions. If the COM port devices are set to be used for logging error messages or debugging, they are not considered in the NOT_PRESENT list. If any task requests for these devices the APP_RES_SET_UNAVAILABLE event is sent to the application.

For an input event, if one or more of the devices or devices for transfer action is NOT_PRESENT and all the other devices are available, the Device Manager sends the output event.

The application gets the output event only if the requested devices are available or they are in the NOT_PRESENT list, i.e., not present on the terminal or malfunctioning. The calls to open these devices in the task will fail when they are NOT_PRESENT.

The application can get the information regarding the entire list of NOT_PRESENT devices at any time by sending an event DEVICE_NOT_PRESENT_EVENT to the Device Manager. The Device Manager responds with the event RESP_DEVICE_NOT_PRESENT_EVENT. This is a 2-byte bitmap as part of its event data. The 2-byte bitmap represents the devices as follows:

The presence of a device can be verified by using the mask definitions.

If devNotPresent is the data returned by devman, devNotPresent, and COMM_5_MASK is true, then the COMM_5 device is not present on the device.

If a particular bit position is set (1), then that device is NOT_PRESENT.

This value is 0x0000 if none of the devices in the requested list, are NOT_PRESENT.

When one or more NOT_PRESENT devices are among the list of devices for the event to be sent the following rules apply:

- If the event is a request event from a task, the Device Manager sends the event if any or all of the devices requested are NOT_PRESENT.
- If the event is an available event and the available device itself is NOT_PRESENT, the event is not sent.
- If the event is an available event and one or more devices requested along with the available device is NOT_PRESENT, the event is sent.

The devices that are not present in the terminal are stored in flexi-record, which can be retrieved from the event data, when application receives RESP_DEVICE_NOT_PRESENT_EVENT as response to DEVICE_NOT_PRESENT_EVENT, from DEVMAN. The flexi-record starts from the 3rd byte in the event data and contains a field with ID as NOT_PRESENT. The type of this field is unsigned long which is of size 32 bits.

Piggyback Events

A task can release devices to the Device Manager and send more than one event using the Device Manager to other tasks using EESL_send_res_release_event() function. The Device Manager takes all the released devices and processes the events *piggybacked*. Piggybacking events ensures that the events are sent to the Device Manager at the same instant as releasing the devices. These events are added to the Device Manager queue and processed.

The limitation with this is that the events piggybacked have their event data size restricted to less than MAX_CUSTOM_EVENT_DATA or MAX_CUSTOM_EVENT_DATA_EX as specified by application in EESL_Initialise() function. This is because all the information (i.e., devReleased, event1, event2) is sent as part event data.

The maximum size allowed for event data of a single piggybacked event from a task which allows event data of size <maximum_allowed_event_data> is:

```
< maximum_allowed_event_data> -
(
     sizeof(short)     //  mask indicating the devices released
    +sizeof(short)   //  number of events piggybacked
    +sizeof(short)   //  indicates the size of first piggybacked event
    +sizeof(short)   //  indicates the size of echodata of first piggybacked
event
    + <echo data from EESL layer, if any>
    +sizeof(short) *2   //part of the event
    + EESL_APP_LOGICAL_NAME_SIZE   //part of the event
)
```

where, <maximum_allowed_event_data> is either MAX_CUSTOM_EVENT_DATA or MAX_CUSTOM_EVENT_DATA_EX.

Echo data is the data internal to the EESL layer which is returned to the sender who has requested this data to be echoed back. For events which are in response to Device Manager originated events, the echo data size is 4 bytes.

If there is no echo data, then:

- MAX_CUSTOM_EVENT_DATA-22 = 278 bytes for tasks allowing event data of size MAX_CUSTOM_EVENT_DATA
- MAX_CUSTOM_EVENT_DATA_EX-22 = 1002 bytes for tasks allowing event data of size MAX_CUSTOM_EVENT_DATA_EX
- For every subsequent event piggybacked:

```
(sizeof(short)  //  indicates the size of piggybacked event
+sizeof(short)  //  indicates the size of echodata of first piggybacked
event
+ echo data from EESL layer if any
+sizeof(short) *2  //part of the event
+ EESL_APP_LOGICAL_NAME_SIZE  //part of the event gets reduced from the
size available for event data of all the events.
```

This is 18 bytes if there is no echo data from EESL layer.

For two piggybacked events, the maximum allowed event data together for all the events is 260 or 984, for three events it is 242 or 966. The maximum number of events that can be piggybacked without event data is 16 if the application has specified MAX_CUSTOM_EVENT_DATA as the maximum event data size. It is 56, if the application has specified MAX_CUSTOM_EVENT_DATA_EX as the maximum event data size in EESL_Initialise() function.

For example, three tasks MAIN, APP1 and APP2 are three co-operating tasks, and task MAIN needs to send APP1_EVENT to task APP1 and APP2_EVENT to task APP2. MAIN is releasing the devices COMM_3, COMM_4 and CONSOLE which were taken in ACT_EVENT. The events APP1_EVENT and APP2_EVENT are listed in the device mapping tables of APP1 and APP2, respectively. APP1_EVENT requires CONSOLE and APP2_EVENT requires COMM_4.

Following is the code snippet without piggybacking:

```
short oEvent = 0;
unsigned char eaData1[MAX_CUSTOM_EVENT_DATA];
unsigned char eaData2[MAX_CUSTOM_EVENT_DATA];
strcpy(eaData1, "APP1EVENT DATA");
strcpy(eaData2, "APP2EVENT DATA"); oEvent= ACT_EVENT;
retVal = EESL_send_event("DEVMAN", RES_COMPLETE_EVENT,
                         (unsigned char*)&oEvent, sizeof(short));
retVal = EESL_send_devman_event("APP1", APP1_EVENT, eaData1,
                                strlen(eaData1)); retVal =
                                EESL_send_devman_event("APP2",
                                APP2_EVENT, eaData2, strlen(eaData2));
```

This required three events to be sent and resulted in synchronization problems if another event arrive requiring devices released before the events APP1_EVENT and APP2_EVENT are sent.

Following is the code snippet using piggybacking facility of Device Manager:

```
// The interface file for EESL_EVENT and the EESL API is eeslapi.h
EESL_EVENT event1;
EESL_EVENT event2; short devReleased;
//for event data unsigned char eaData1[MAX_CUSTOM_EVENT_DATA]; unsigned
char eaData2[MAX_CUSTOM_EVENT_DATA];
// devices being released are COMM_3, COMM_4, CONSOLE devReleased=
COMM_3_MASK | COMM_4_MASK | CONSOLE_MASK;
// initialize the event objects with the event id, the destination task
and the event data event1.customEESLEvent = APP1_EVENT;
strcpy(event1.destinationTask, "APP1");
strcpy(eaData1, "APP1EVENT DATA");
event1.EESLEventData = eaData1;
event1.evtDataLength = strlen(eaData1);
strcpy(event2.destinationTask, "APP2");
event2.customEESLEvent = APP2_EVENT;
strcpy(eaData2, "APP2EVENT DATA");
event2.EESLEventData = eaData2;
event2.evtDataLength = strlen(eaData2);
// the events APP1_EVENT and APP2_EVENT are listed in the device
// mapping tables of APP1 and APP2
EESL_send_res_release_event(devReleased,&event1,&event2,NULL);
```

HOTKEY Configuration

The HOTKEY is the key used to return to VMAC from a selected application. The Device Manager is the owner of the HOTKEY. This key is configurable. The following items apply when using HOTKEYs:

- To determine the HOTKEY, the Device Manager looks for #HOTKEY in the CONFIG.SYS of GID1. If not found, the HOTKEY defaults to the asterisk (*) key.
- The #HOTKEY value specifies the keycode to use as the HOTKEY. The key code value must be specified as a hex constant specified in C language notation. For example, #HOTKEY=\x2a sets the HOTKEY to the asterisk key.

NOTE

If #HOTKEY specifies an invalid keycode value for the terminal, the behavior is undefined.

Applications must enable the HOTKEY when application switching is permitted. Otherwise, the application should disable the HOTKEY. The OS provides APIs for enabling (enable_hot_key) and disabling (disable_hot_key) the HOTKEY. When the HOTKEY is pressed the application gets a deactivate event from the OS (EVT_DEACTIVATE). In the handler for this event, the application should send RES_COMPLETE_EVENTs to the Device Manager to return all allocated devices.

Using Device Manager

The following are guidelines for using the Device Manager:

1. Decide on the devices required by the task.
2. Decide on when these devices are required.
3. Define events to get these devices.
4. Decide on the priority of these events.
5. Define these events, the output events, and the list of devices required in the device mapping resource.
6. Handle the output event of `APPLICATION_INIT_EVENT` if this has been requested by the application by putting this in the device mapping table.
7. Send the request events to the Device Manager when these devices are needed.
8. Handle the response event that is the output event from the Device Manager to use the device.
9. Send the `RES_COMPLETE_EVENT` to the Device Manager to release the devices obtained in the response event after they are done with.
10. Handle `APP_RES_SET_UNAVAILABLE` event which informs that the device requested was not available.
11. Handle <device>`_REQUEST_EVENT`s if devices are obtained in MEDIUM or LOW priority.
12. Send `RES_REJECT_EVENT` to the Device Manager if an unknown event is sent by DEVMAN.
13. Handle the output events of any <device>`_AVAILABLE_EVENT` if this has been requested by the application by putting this in the device mapping table.
14. Disable the <device>`_AVAILABLE_EVENT`s when they are not required to be sent by sending `DISABLE_EVENT_ACTION`. And enable them when they are required to be sent by sending `ENABLE_EVENT_ACTION`.
15. Handle `EVENT_NOT_HANDLED` event which informs that the event was not found in the device mapping table, sending the event to the destination failed the event has been disabled by the receiver.
16. Use EESL_send_res_release_event() function to release devices and piggyback request events and/or events to other tasks using the Device Manager.

Device Manager Best Practices

- The Device Manager is the default owner of all the devices.
- All the device requirements of a task must be specified before hand in the device mapping table provided to the Device Manager.

- The output event is sent to a task only if all the devices are available or are not present on the terminal. Refer to the Handling Devices Not Present on the Terminal section for more details.
- The Device Manager makes no provision for queuing devices required by more than one application.
- The tasks must return the devices after it has completed using them.
- The device can be returned using:

  <device>_ RELEASED_EVENT - this is ideal when only one device is being released.

  or

  RES_COMPLETE_EVENT - this is ideal when more than one device is being released, the devices released are those that were allocated by an output event by the Device Manager. This output event is sent as event data.
- The devices taken on a low priority must be released if requested.
- If any of COMM_1 or COMM_2 port is being used in the application, then do not set this port as DEBUGPORT or LOGPORT.
- Use EESL_send_devman_event() function to send an event to a task using the Device Manager. This could be used even for request events.
- Use EESL_send_res_release_event() function to release devices and piggyback request events and/or events to other tasks using the Device Manager. Refer to the Piggyback Events for more details.

APPLICATION_INIT_EVENT

This is one of the events, which is sent by the Device Manager to the application at the time of startup. This event is sent only if the task has listed the event in the device mapping table, and the devices requested are available with the Device Manager.

If the application registers to get an `APPLICATION_INIT_EVENT` and has COM2 or COM3 or both the devices specified in the device mapping table, the Device Manager will close both COM2 and COM3, and send the output event to the respective application.

NOTE

It is recommended to use `APPLICATION_INIT_EVENT` macro in the `<<Application Logical Name>>.rck` file only once.

Device Available Events

The device available events are sent when a device is available and is not requested by any other application. The available event must be listed in the device mapping table of that application.

Device Request Events

The device request event is sent when a device is with an event that has a priority lower than that of the event requesting the device. The request events are not listed in the device mapping table and are sent only by Device Manager.

Device RES_COMPLETE_EVENT

The application sends this event to the Device Manager to release all devices that the Device Manager allocated. The application must send the event ID for `RES_COMPLETE_EVENT` in the event data. It is the responsibility of the application to close all the devices before sending this event, else the event fails. If event ID specified in the `RES_COMPLETE_EVENT` has any of COM2 or COM3 or both COM2 and COM3 it has to close both the devices for the event to be successful.

Device Release Events

The device release events are sent by the application to the Device Manager when a device is released by the application either as a response to a device request by the Device Manager or as a partial release of devices or releasing the devices after the completion of task.

APP_RES_SET_UNAVAILABLE

This is an acknowledgment event, which is sent by Device Manager to the application requested when the resource is not available.

For example, APP1 and APP2 are two applications that are downloaded. When APP1 has already acquired COM2 on high priority and if APP2 requests for COM2, the Device Manager will send an `APP_RES_SET_UNAVAILABLE` acknowledgement to APP2.

Resource Manager

This section lists the API calls provided by RM (Resource Manager). These API calls help to access the resources generated by the resource compiler. Refer to the Verix V Tools *Readme* for more details.

- Initialization
  - `RM_Start()`
  - `RM_ShutDown()`
- Dynamically opening/closing resource file
  - `RM_OpenFile()`
  - `RM_CloseFile()`
- Loading resources
  - `RM_LoadResFromFile()`
  - `RM_LoadRes()`
  - `RM_LoadPartialRes()`
- Getting handles to resources
  - `RM_GetResOnTypeId()`
  - `RM_GetCurrResOnTypeId()`
- Iterating over a resource
  - `RM_GetResTypeIdOnIndex()`
  - `RM_GetCurrResTypeIdOnIndex()`
- Iterating over resources based on resource type
  - `RM_GetResOnTypeIndex()`
  - `RM_GetCurrResTypeIdOnIndex()`
- Replacing a resource
  - `RM_ReplaceRes()`
  - `RM_ReplacePartialRes()`
- Counting Resources
  - `RM_GetNumResOnType()`
  - `RM_GetCurrNumResOnType()`
  - `RM_GetNumRes()`

- `RM_GetCurrNumRes()`
- `RM_GetNumResInFile()`

- Getting resource information
  - `RM_GetResInfo()`
- Accessing member of RHANDLE
  - `RM_GetFNameFromHandle()`
  - `RM_GetPosFromHandle()`
  - `RM_GetResSzFromHandle()`

RM_CloseFile()

Removes the resource file from the resource chain.

Prototype

```
#include <k2api.h>
short RM_CloseFile(char *fName);
```

Parameters

`fName`	The base name of the resource file.

Return Values

Success:	SUCCESS (0)
Failure:	E_RM_NOTSTARTED: RM not initialized with RM_Start() function.
	E_RM_NOTOPEN: Specified resource file not appended to the resource chain using RM_OpenFile() function.
	E_RM_BADFILE: Resource file does not exist.

NOTE

The resource file extension is always "`.res`".

RM_GetCurrNumRes()

Returns the total number of resource instantiations in the resource file at the top of the resource chain.

Prototype

```
#include <k2api.h>
short RM_GetCurrNumRes(void);
```

Return Values

Success:	Number of resource instantiations.
Failure	E_RM_NOTSTARTED: RM not initialized.

RM_GetCurrNumResOnType()

Returns the number of resource instantiations of the specified resource template ID, which is present at the top of the resource chain.

Prototype

```
#include <k2api.h>
short RM_GetCurrNumResOnType(RESTYPE rt);
```

Parameters

`rt`	The resource template ID.

Return Values

Success:	The number of resource instantiations belonging to the type `rt` present on top of the resource chain.
Failure:	E_RM_NOTSTARTED: RM not started.
	E_RM_CHNEMPTY: No resource instantiations are in the resource chain.

RM_GetCurrResOnTypeId()

Retrieves the handle to a resource instantiation whose resource template and resource ID are specified. The specified resource instantiation is searched in the resource file at the top of the resource chain. For example, if the resource chain has R3-R2-R1, the specified resource instantiation is searched only in R3.

Prototype

```
#include <k2api.h>
short RM_GetCurrResOnTypeId(RESTYPE rt, unsigned short resID,
RHANDLE *handle);
```

Parameters

`rt`	The resource template ID.
`resID`	The resource ID.
`handle`	Contains the handle to the resource instantiation.

Return Values

Success:	SUCCESS (0) with `handle`, which contains the handle to the required resource instantiation.
Failure:	E_RM_CHNEMPTY: Empty resource chain.
	E_RM_BADRES: Invalid resource template ID or resource ID.
	E_RM_NOTSTARTED: RM not started.
	E_RM_BADHANDLE: Invalid handle.

RM_GetCurrResOnTypeIndex()

Obtains the handle to the index; the resource instantiation whose resource template ID is specified. The handle is obtained for the specified resource instantiation in the resource file at the top of the resource chain. For example, if the resource chain consists of the files R3-R2-R1, the search is carried out only in the file R3.

Prototype

```
#include <k2api.h>
short RM_GetCurrResOnTypeIndex(RESTYPE rt, unsigned short index,
RHANDLE *handle);
```

Parameters

`rt`	The resource template ID.
`index`	The index number of the resource instantiation in the resource file at the top of the resource chain.
`handle`	The handle to the resource instantiation.

Return Values

Success: SUCCESS (0) with `handle`: The handle to the required resource instantiation.

Failure: E_RM_CHNEMPTY: Empty resource chain.

E_RM_BADINDEX: Invalid resource index.

E_RM_BADRES: Invalid resource template ID.

E_RM_NOTSTARTED: RM not started.

E_RM_BADHANDLE: Invalid handle.

RM_GetCurrResTypeIdOnIndex()

Retrieves the resource template ID and resource instantiation ID given the index. This call is applicable only to the resource file at the top of the resource chain. For example, if the resource chain is R3-R2-R1 and R3, R2, and R1 contain 3, 4, and 5 resource instantiations, respectively. Then, `index` = 2 implies the second resource instantiation is in the resource file R3. An index value of 6 is incorrect because the resource specification file R3, which is on the top of the resource chain, has only 3 resources.

Prototype

```
#include <k2api.h>
short RM_GetCurrResTypeIdOnIndex(unsigned short index, RESTYPE* rt,
unsigned short *resID);
```

Parameters

`index`	The index number of the resource instantiation in the resource specification files at the top of the resource chain.
`rt`	The resource template ID.
`resID`	The resource ID.

Return Values

Success: SUCCESS (0), with the following values:

- `rt`: the resource template ID.
- `resID`: the resource ID.

Failure: E_RM_NOTSTARTED: RM not initialized.

E_RM_CHNEMPTY: Resource chain empty.

E_RM_BADINDEX: Invalid resource index.

E_RM_BADRESTYPE: Invalid resource template ID.

E_RM_BADRES: Invalid resource ID.

RM_GetFNameFromHandle()

Copies the resource specification file name from `handle` to `fname`.

Prototype

```
#include <k2api.h>
short RM_GetFNameFromHandle(RHANDLE* handle, char* fname,
unsigned short length);
```

Parameters

`handle`	A pointer to the RHANDLE structure from which the resource specification file name is obtained.
`fname`	Filename of the resource specification is copied from `handle` and passed to this buffer.
`length`	The size of the `fname` buffer. Only `length` number of bytes is copied to `fname`.

Return Values

Success:	SUCCESS (0)
Failure	`E_RM_BADARGUMENT`: `fname` is null, `handle` is null, or `length` is zero or greater than the size of `fname`.

RM_GetNumRes()

Returns the total number of resource instantiations in the entire resource chain.

Prototype

```
#include <k2api.h>
short RM_GetNumRes(void);
```

Return Values

Success:	Number of resource instantiations.
Failure:	`E_RM_NOTSTARTED`: RM not initialized.

RM_GetNumResInFile()

Retrieves the total number of resource instantiations in a resource specification file. Use this information to allocate a proper buffer.

Prototype

```
#include <k2api.h>
short RM_GetNumResInFile(char *fName, unsigned short *noResource);
```

Parameters

`fName`	The base name of the resource file.
`noResource`	The number of resource instantiations in the file filled by the API.

Return Values

Success:	SUCCESS
Failure:	E_RM_BADFILE: Resource file not present.
	E_RM_RESERROR: File error.
	E_RM_BADARGUMENT: Invalid argument.

RM_GetNumResOnType()

Returns the number of resource instantiations for the specified resource template ID present in the entire resource chain.

Prototype

```
#include <k2api.h>
short RM_GetNumResOnType(RESTYPE rt);
```

Parameters

`rt`	The resource template ID.

Return Values

Success:	Returns the number of resource instantiations belonging to the specified resource template ID in the entire resource chain.
Failure:	`E_RM_NOTSTARTED`: RM not started.
	`E_RM_CHNEMPTY`: No resources are instantiated in the resource chain.

RM_GetPosFromHandle()

Copies the position of the resource stored in `handle` to `posn`.

Prototype

```
#include <k2api.h>
short RM_GetPosFromHandle(RHANDLE* handle, unsigned short* posn);
```

Parameters

`handle`	A pointer to the `RHANDLE` structure from which the resource position is obtained.
`posn`	The resource position is copied from `handle` and passed to this parameter.

Return Values

Success:	SUCCESS (0)
Failure:	E_RM_BADARGUMENT: `posn` is NULL or `handle` is NULL.

RM_GetResIDFromHandle()

Copies the resource ID from `handle` to `resID`.

Prototype

```
#include <k2api.h>
short RM_GetResIDFromHandle(RHANDLE* handle, unsigned short* resID);
```

Parameters

`handle`	A pointer to the `RHANDLE` structure from which the resource ID is obtained.
`resID`	Contains the resource ID copied from `handle` and passed to this parameter.

Return Values

Success:	SUCCESS (0)
Failure:	`E_RM_BADARGUMENT`: `resID` is null or `handle` is null.

RM_GetResInfo()

Retrieves information about a particular resource instantiation. Use RM_GetResInfo() function to determine the exact size of a resource to allocate the required buffer size.

Prototype

```
#include <k2api.h>
short RM_GetResInfo(char *fName, RESTYPE rt, unsigned short resID,
RESINFO *rInfo);
```

Parameters

`fName` The base name of the resource specification file.

`rt` The resource template ID. The information of the resource with the given `resId` and resource template ID `rt` will be stored in `rInfo`.

`resID` The resource ID.

`rInfo` A pointer to the RESINFO structure. The structure of RESINFO is as follows:

```
typedef struct resinfo
{
RESTYPE resType;
unsigned char property;
unsigned short size;
}RESINFO;
```

Return Values

Success: SUCCESS

Failure: `E_RM_BADFILE`: The resource specification file not present.

`E_RM_RESERROR`: File error.

`E_RM_BADRES`: Resource ID not found in the resource specification file.

RM_GetResOnTypeId()

Retrieves the handle to a resource instantiation whose resource template ID and resource ID have been specified. This resource is searched for in the entire resource chain, such as, in all resource specification files opened using RM_OpenFile() function.

Prototype

```
#include <k2api.h>
short RM_GetResOnTypeId(RESTYPE rt, unsigned short resID, RHANDLE
*handle);
```

Parameters

`rt`	The resource template ID of the required resource instantiation.
`resID`	The resource ID of the required resource instantiation.
`handle`	The handle to the resource instantiation.

Return Values

Success: SUCCESS (0) with `handle`, which contains the handle to the resource instantiation.

Failure: `E_RM_CHNEMPTY`: Empty resource chain.

`E_RM_BADRES`: Invalid resource template ID or resource ID.

`E_RM_NOTSTARTED`: RM not started.

`E_RM_BADHANDLE`: Invalid handle.

RM_GetResOnTypeIndex()

Retrieves the handle to the resource instantiation specified and is the indexth resource in the entire resource chain (i.e., in all the resource files opened using RM_OpenFile() function).

Suppose there are two separate resource specification files with resource instantiations for the same template and neither of the two files have the index number of resource instantiations in individual files. Then, if the index is less than the sum of the number of resource instantiations in each file, this call returns a success. The handle contains the (index - number of resources in first file)th resource in the second file. This is because the entire resource chain is being searched.

Prototype

```
#include <k2api.h>
short RM_GetResOnTypeIndex(RESTYPE rt, unsigned short index,
RHANDLE *handle);
```

Parameters

`rt`	The resource template ID.
`index`	The index number of the resource instantiation in the resource specification files in the entire resource chain.
`handle`	The handle to the resource instantiation.

Return Values

Success:	SUCCESS (0) with `handle`: The handle to the required resource instantiation.
Failure:	`E_RM_CHNEMPTY`: Empty resource chain.
	`E_RM_BADINDEX`: Invalid resource index.
	`E_RM_BADRES`: Invalid resource template ID or resource ID.
	`E_RM_NOTSTARTED`: RM not started.
	`E_RM_BADHANDLE`: Invalid handle.

RM_GetResSzFromHandle()

Copies the resource size from `handle` to `resSz`.

Prototype

```
#include <k2api.h>
short RM_GetResSzFromHandle(RHANDLE* handle, unsigned short* resSz);
```

Parameters

`handle`	A pointer to the RHANDLE structure from which the resource size is obtained.
`resSz`	The size of the resource is copied from `handle` and passed to this parameter.

Return Values

Success:	SUCCESS (0)
Failure	`E_RM_BADARGUMENT`: `resSz` is null or `handle` is null.

RM_GetResTypeIdOnIndex()

Retrieves the resource template ID and resource ID of a resource with the index of the resource. This function retrieves the resource instantiation for the specified index in the entire resource chain.

For example, if the resource chain is R3-R2-R1 and R3, R2, and R1 contain 3, 4, and 5 resource instantiations, respectively. In total, there are 12 resource instantiations. `index` = 6 implies that the third resource instantiation is in the resource specification file R2. This is because it is the sixth resource instantiation in the entire chain.

Prototype

```
#include <k2api.h>
short RM_GetResTypeIdOnIndex(unsigned short index, RESTYPE* rt,
unsigned short *resID);
```

Parameters

`index`	The index number of the resource in the resource chain.
`rt`	The resource template ID.
`resID`	The resource ID.

Return Values

Success: SUCCESS (0), with the following values:

- `rt`: the resource template ID.
- `resID`: the resource ID.

Failure: E_RM_NOTSTARTED: RM not initialized.

E_RM_CHNEMPTY: Empty resource chain.

E_RM_BADINDEX: Invalid resource index.

E_RM_BADRESTYPE: Invalid resource template ID.

E_RM_BADRES: Invalid resource ID.

RM_LoadPartialRes()

Partially loads resources. RM_LoadResFromFile() and RM_LoadRes() function load the entire resource. `RM_LoadPartialRes()` takes two additional arguments: The start position and length. `RM_LoadPartialRes()` function loads `len` bytes of the resource from the `startPosn` byte of the resource.

Prototype

```
#include <k2api.h>
short RM_LoadPartialRes(RHANDLE *handle, char*buff,
unsigned short startPosn, unsigned short len);
```

Parameters

`handle`	Contains the handle to the resource to load.
`buff`	User-allocated buffer filled by RM.
`startPosn`	The position of the resource from the start of the resource file.
`len`	The number of bytes to load.

Return Values

Success:	SUCCESS (0) and the buffer filled with the resource.
Failure:	E_RM_BADHANDLE: Invalid resource handle.
	E_RM_BADARGUMENT: Invalid buffer.
	E_RM_BADFILE: Resource file not found.
	E_RM_RESERROR: Error encountered while reading the file.
	E_RM_BUFSZ: Size of the passed buffer is less than the resource size.

RM_LoadRes()

Loads the resource whose handle is supplied as the parameter.

Prototype

```
#include <k2api.h>
short RM_LoadRes(RHANDLE *handle, unsigned int bufSz, char *buff);
```

Parameters

`handle`	Contains the handle to the resource to load.
`bufSz`	The size of the allocated buffer.
`buff`	User-allocated buffer filled by RM.

Return Values

Success:	SUCCESS (0) and the buffer filled with the resource.
Failure:	E_RM_BADHANDLE: Invalid resource handle.
	E_RM_RESERROR: Error encountered while reading the file.
	E_RM_BUFSZ: The size of the allocated buffer is less than the resource size.
	E_RM_BADARGUMENT: Invalid argument.
	E_RM_BADFILE: Resource file does not exist.

RM_LoadResFromFile()

Copies a resource from the resource file to `buff`.

NOTE

You must ensure that the buffer is large enough to hold the resource value.

Prototype

```
#include <k2api.h>
short RM_LoadResFromFile(char *fName, RESTYPE rt, unsigned short resId,
unsigned int bufSz, char* buff);
```

Parameters

`fName`	The base name of the NULL-terminated resource file.
`rt`	The template of the resource to load. The resource with the given `resId` and with resource template ID `rt` will be stored in the buffer.
`resId`	The resource ID of the resource to load.
`bufSz`	The size of the passed buffer.
`buff`	User-allocated buffer to fill with the resource.

Return Values

Success:	SUCCESS (0) and the buffer filled with the resource.
Failure:	E_RM_BADFILE: Resource file does not exist.
	E_RM_RESERROR: Error encountered while reading the file.
	E_RM_BUFSZ: The size of the allocated buffer is less than the resource size.
	E_RM_BADRES: Invalid resource template ID or ID.
	E_RM_BADARGUMENT: Invalid argument.

RM_OpenFile()

Adds the specified resource file to the resource chain. This added file becomes the new “top” of the resource chain. For example, if the chain initially consisted of R2-R1. R3 is the new open file. The resource chain now has R3-R2-R1. The order for searching resources is R3-R2-R1.

Prototype

```
#include <k2api.h>
short RM_OpenFile(char *fName);
```

Parameters

`fName`	The base name of the resource file. Maximum of 8 characters. If the file is in terminal flash or in GID15, the file name can be a maximum 8 characters, excluding '/' (For GID15), and/or “`f:`” (for flash).

Return Values

Success:	SUCCESS(0)
Failure:	E_RM_NOTSTARTED: RM not initialized with RM_Start() function. E_RM_BADFILE: Specified resource file does not exist. E_RM_LOWMEM: No memory to load resource chain.

NOTE

The resource file extension is always “`.res`”.

RM_ReplacePartialRes()

Partially replaces the resource pointed to by `handle`. The resource is replaced with the contents of the buffer, with the starting position and length specified.

NOTE

The sum of the starting position and the length cannot exceed the resource size.

Prototype

```
#include <k2api.h>
short RM_ReplacePartialRes(RHANDLE *handle, char *buff,
unsigned short startPosn, unsigned short len);
```

Parameters

`handle`	The resource handle of the resource to replace.
`buff`	User-allocated buffer to replace.
`startPosn`	The start position of the resource to replace.
`len`	The number of bytes to replace.

Return Values

Success:	SUCCESS (0)
Failure:	E_RM_BADHANDLE: Invalid resource handle.
	E_RM_BADARGUMENT: Invalid buffer.
	E_RM_BADFILE: Resource file not found.
	E_RM_RESERROR: Error encountered while reading the file.
	E_RM_BUFSZ: The size of the passed buffer is less than the resource size.

RM_ReplaceRes()

Replaces the resource pointed to by `handle` with the contents of the buffer whose size is in `bufSz`.

NOTE

The resource must be entirely replaced. Partial replacement of a resource results in an error.

Prototype

```
#include <k2api.h>
short RM_ReplaceRes(RHANDLE *handle, unsigned int bufSz, char *buff);
```

Parameters

`handle`	Contains the handle to the resource to load.
`bufSz`	The size of the allocated buffer in bytes.
`buff`	User-allocated buffer to replace.

Return Values

Success:	SUCCESS
Failure:	E_RM_NOTSTARTED: RM not started.
	E_RM_BADHANDLE: Invalid resource handle.
	E_RM_BADFILE: Resource file not present.
	E_RM_BUFSZ: The size of the allocated buffer is not equal to the size of the resource to replace.

RM_ShutDown()

Shuts down RM and releases memory allocated to the resource chain.

NOTE

Call this function only once (before you exit the application).

Prototype

```
#include <k2api.h>
void RM_ShutDown(void);
```

RM_Start()

Starts the resource manager.

NOTE

This API must be called before any other RM API is called.

Prototype

```
#include <k2api.h>
short RM_Start(char MaxNoRes);
```

Parameters

`MaxNoRes`	The maximum number of resource files that the application can use. This value should not exceed 50.

Return Values

Failure:	E_RM_2MANYFILES: Number of resource files is more than set in `MaxNoRes`.

RM Error Codes

The following are error messages you may encounter while compiling the resources:

“Cannot evaluate expression.”

“Duplicate element name!”

“Duplicate enum name!”

“ERROR! Invalid Syntax!!”

“ERROR! No files to operate on!“

“Error encountered! Aborting compilation!”

“ERROR in instantiation file! Aborting compilation!!”

“ERROR in template file! Aborting compilation!!”

“ERROR! Aborting compilation!”

“ERROR! Can't create file[%s] \n", toFile);”

“ERROR! Duplicate resource IDs["<<rid1<<"].”

“ERROR! Duplicate resource names ["<<rName1<<"].”

“ERROR! Duplicate template IDs["<<tid1<<"].”

“ERROR! Duplicate template names ["<<tName1<<"].”

“ERROR! Incorrect number of elements in array["<< tNode->getName()<<"] of resource["<<_rName <<"] of template["<<_tName<<"].”

“ERROR! Incorrect number of elements in CASE statement[" << caseTNode->getName() << "].”

“ERROR! Instantiation file base name too long!!”

“ERROR! Instantiation file does not exist!!”

“ERROR! Invalid case[" << caseName << "]in resource[" << _rName << "].”

“ERROR! Invalid Option!”

“ERROR! Invalid value for element [" << tNode->getName()<< "] in resource [" << _rName << "].”

“ERROR! Number of sub-arrays in fixed size array[" << tNode->getName() << "] of resource [" << _rName << "] incorrect.”

“ERROR! Resource [R1] and template [T1] do not match.”

“ERROR! Resource creation abandoned!”

“ERROR! Resource source file does not exist!!”

"ERROR! Output file does not exist!!"

"ERROR! Specify either one of resource source or instantiation file!"

"ERROR! Template file base name too long!!"

"ERROR! Template file does not exist!!"

"ERROR! Template["<<tName<<"] specified in instantiation NOT present!"

"ERROR! Too many bits in element"<< "[" <<name<< "]."

"ERROR! Value for element."

"Illegal array size!"

"Illegal value for unsigned enum in resource element!"

"Illegal value for unsigned resource element!"

"Internal error! Argument list exceeds 1024 bytes."

"Internal error! File or path not found."

"Internal error! Mode argument is invalid."

"Internal error! Not enough memory is available to execute process."

"Internal error! Specified file is not executable or has invalid executable-file format."

"Invalid element name!"

"Invalid File Name!"

"Invalid Hex String Value!"

"Invalid resource name!"

"Invalid Value!"

"Parse Error in file[%s]! Invalid token [%s] at line %d offset %d\n"

"String value too long!"

"Token Error!"

"Invalid template name!"

"Warning! The cstring resource is being truncated as it exceeds the array size."

"Warning! The pstring resource is being truncated as it exceeds the array size."

"Warning! The pcstring resource is being truncated as it exceeds the array size."

Table 5 lists error codes defined by the Resource Manager and their relevant explanations.

Table 5 Resource Manager Error Codes

Error Codes	Description
E_RM_BADFILE	Resource file does not exist.
E_RM_BADRES	Invalid resource ID.
E_RM_BUFSZ	Resource buffer size too small.
E_RM_RESERROR	Resource file error.
E_RM_NOTSTARTED	Resource manager not initialized.
E_RM_NOTOPEN	Resource file not opened.
E_RM_LOWMEM	No memory to load resource chain.
E_RM_MAXRES	Maximum number of resource files opened.
E_RM_FILEERR	Resource file access error.
E_RM_CHNEMPTY	Resource chain empty.
E_RM_BADINDEX	Invalid index.
E_RM_BADHANDLE	Invalid handle.
E_RM_2MANYFILES	Number of resource files more than MaxNoRes.
E_RM_BADRESTYPE	Invalid resource type.
E_RM_BADARGUMENT	Invalid argument.
E_RM_BADVERSION	Incompatible version of resource.

CHAPTER 7

FrontEnd

The FrontEnd is a service application of VMAC that provides menus to invoke applications manually. FrontEnd is provided as a Verix eVo executable and runs in GID1 along with other core VMAC components.

This chapter provides information about features, configuration and customization options.

Features

FrontEnd provides a simple application selection environment that VeriFone and VIP programmers can use with other independently developed applications on Verix eVo OS compatible terminals. When running in the terminal, FrontEnd provides an easy to use user interface for selecting applications.

FrontEnd features include:

- Automatic creation of an application selection menu listing all the applications registered with IMM (listed in the `default.INI` file). The `.INI` file can contain subtasks that are controlled by the running application.
- If only one application is loaded in the VMAC environment, the application is automatically executed. In this case no menu selection will be displayed. If sending the activation event fails then FrontEnd retries activating the task automatically for the seconds set in the `#ACTIVATE_RETRY_TIMER CONFIG.SYS` variable. After this, FrontEnd displays a failure screen and asks for a key press to retry activating.
- Application activation happens by selecting an application from the displayed application list and pressing one of the four function keys to the right of the screen.
- Facility to display more than one menu item for a task and thus more than one paths of activation of a task from FrontEnd.
- Configurable font file size to display the list of applications. The supported fonts include 6x8, 8x16, and 16x16.
- Configurable menu option names. Text for each menu item to identify each application.
- Support for configurable HOTKEY to return the control to VMAC from the application.
- Transfer of terminal control, upon selecting an application from the menu, to the selected application until a HOTKEY press or using a menu option in the application.

- Customizable screen saver display when FrontEnd is idle.
- Configurable menu items. Application names displayed in FrontEnd may be reordered.
- Support for a secondary FrontEnd menu.
- Support for “Latent” applications.
- Display and Printing of `CONFIG.SYS` variables.
- Support for a Dynamic Menu. The application can add and delete menu items at runtime by calling `Add_Menu_Item()` and `Remove_Menu_Item()`.

Table 6 lists the configuration options that are common to VX520 and VX680 terminals. Refer to the *Verix eVo Multi-App Conductor for Touch screen Terminals Users Guide* for specific configuration options about VX680 terminal.

Table 6 Configuration Options

Supported Terminal	Configuration Options	Parameters	Location
VX520	Menu item text	MenuText	Ins File
	Font size selection	#FONTSIZE	Config Variable
	Screen saver enabling/ disabling	Screen Saver	Config Variable
	Modification of the screen saver	Screen Saver	Ssaver.ini
	Activation retry settings	#ACTIVATION_RETRY _TIMER	Config Variable
	Enabling and disabling clock display	FECLOCK	Config Variable
	Configurable application activation time	SWITCHTIME	Config Variable
	Date display mode	FEDATE	Config Variable/ Femessag.ini
	Time display mode	FETIME	Config variable/ Femessag.ini
	Reordering menu items	Menu Order	Config Variable

Table 6 Configuration Options (continued)

Supported Terminal	Configuration Options	Parameters	Location
	Secondary FrontEnd menu	OTHERMENU	Config Variable
	Latent applications	LATENT	Ins File
	Application password	APPNAME_PASSW	Config Variable
VX680	Menu item text	MenuText	Ins File
	Screen saver enabling/ disabling	Screen Saver	XML File
	Enabling and disabling clock display	Date, Time	XML File
	Reordering menu items	Menu Order	Ins File
	Secondary FrontEnd menu	OTHERMENU	Ins File
	Latent applications	LATENTKEY	Ins File
	Application password	PASSWORD	Ins File

Menu Item Text

The menu item to display for the application in the FrontEnd menu is obtained from the `.INS` file of the application. The file prefix is the logical name of the application as specified in the `default.INI` file. The file must be loaded into the same directory as the associated application, not necessarily on the same drive (RAM or flash).

The <application logical name>`.ins` is of the following format:

```
[ActivateEvent]
EVENT1=<Activate event id 1>
EVENT2=<Activate event id 2>
.[MenuText]
TEXT1=<Text for FrontEnd Menu 1>
TEXT2=<Text for FrontEnd Menu 2>
```

For example:

```
[ActivateEvent]
EVENT1=15000
```

```
EVENT2=15001
[MenuText]
TEXT1=PW SALE
TEXT2=PW OFFLINE SALE
```

The first section contains a list of events that activates the task. The keys are EVENT1, EVENT2, and so on, and the values are the event IDs that activate the application. This ensures that FrontEnd displays more than one item on its menu for a task. The number items displayed is the number of events listed in the `.INS` file.

The second section contains the entries done with the keys TEXT1, TEXT2, and so on. The strings given for the keys display in the FrontEnd menu. Text is given for all the events listed in the `[ActivateEvent]` section. If the text for any event does not exist, the text used is the application logical name appended with index of the event for which the text does not exist.

For example, if the application logical name is APPNAME for which TEXT1 and TEXT3 are not provided, the menu texts displayed are:

- APPNAME
- TEXT2
- APPNAME

If the logical name of the application is longer than the text that can be displayed on the FrontEnd menu, then the text is truncated and then the index is appended. For the above example, if the font size is large (for example, 16x16), menu text can only be 7 characters long. The menu texts displayed are:

- APPNAM
- TEXT2
- APPNAM

NOTE

If the events are not correctly specified, i.e., after EVENT4, EVENT6 is specified, then all events up to EVENT4 display on the menu.

If there is only one task and it has two activate events, then FrontEnd displays the menu with those two events. This situation is not considered as a single application situation, and the disable HOTKEY event is not sent to the Device Manager.

The `.INS` file for the application may contain the text from a Big font file (for example, font files with a 2-byte character index that containing more than 256 characters). The text must be the 2-byte index string from the font file set in the `#FONTNAME CONFIG.SYS` variable in FrontEnd. The text can be provided as follows:

For example, for a normal font file, the text in the `application .INS` file is:

```
TEXT1=APPLICATION ONE
```

and for a Big font file:

```
TEXT1=\x01\x41\x01\x50\x01\x50\x01\x4c\x01\x49\x01\x43\x01\x41\x01\x54\yx
01\x49\x01\x4f\x01\x4e\x00\x20\x01\x4f\x01\x4e\x01\x45
```

Selecting Font Size

FrontEnd supports the following three font sizes:

- 6x8: Default font file FEENSML.VFT (provided)
- 8x16: Default font file FEENMED.VFT (provided)
- 16x16: Default font file FEENLRG.VFT (provided)

The font size and the font file can be selected by setting the `CONFIG.SYS` environment variables FONTSIZE and FONTNAME in GID1. FONTNAME contains both the filename and the path for the font file. The path is in the form: <Drive>:<current gid or Gid 15>.

GID15 is specified by a slash (\), and the current GID is specified by not having anything for the GID information in the path.

For a 6x8 display using the font file `FEENSML.VFT` downloaded to GID1 in flash:

```
#FONTSIZE=S
#FONTNAME=F:FEENSML.VFT
```

For a 8x16 display using the font file `FEENMED.VFT` downloaded to GID1 in RAM:

```
#FONTSIZE=M
#FONTNAME=FEENMED.VFT
```

For a 16x16 display using the font file `FEENLRG.VFT` downloaded to GID15 in flash:

```
#FONTSIZE=L
#FONTNAME=F:\FEENLRG.VFT
```

NOTE

- All labels display in the selected font size. If the text entered into the <application logical name>`.INS` file is greater than the number of characters that can be displayed per line for a specific font, then the menu text is truncated.
- VMAC FrontEnd will be aligned correctly only when `#FONTNAME` variable is set to `F:FEENSML.VFT` variable, in the absence of `#FONTSIZE` config variable.

Enabling/Disabling the Screen Saver

If the FrontEnd remains idle for some time, it displays an informational screen. This screen saver feature is conditionally built into the FrontEnd screen. The screen saver is customizable using the `ssaver.ini` file. A configurable option is given in the `CONFIG.SYS` file to enable and disable the screen saver. The screen saver displays only when the variable SSAVER variable in the `CONFIG.SYS` is set to 1. Any value to this variable other than 1 disables the screen saver. This allows the users to disable or enable the screen saver at runtime though the `ssaver.ini` screen saver file is downloaded into the terminal.

To disable the screen server, do not download the `ssaver.ini` file. When the `ssaver.ini` is not downloaded, the SSAVER variable entry in the `CONFIG.SYS` file is ignored. This feature is applicable only for VX520 terminals.

Modifying the Screen Saver Display

To modify the appearance of the informational displays, edit the `ssaver.ini` file, and change the screen saver resource.

The screen saver can be specified as a combination of graphics and text. The different lines form the different sections of the screen saver resource. Each section gives the details of the row, column and font file. Each line on the screen can be displayed as a combination of a graphic and text. The different sections are identified as LINE1, LINE2, and so on, for text display and GRAPHIC0, GRAPHIC1, and so on, for graphical display.

The text resource consists of the fields TEXT, FONT, COLUMN, and ROW, as follows:

```
[LINE<n>]
;where, n is the line number
TEXT=<text to display>
FONT=<font to display the text in>
COLLUMN=<column on the screen to start the text at>
ROW=<row on screen to start the text at>
```

Lines with ';' as the beginning character are considered comments.

Similarly, the graphic item has the following fields

```
[GRAPHIC<n>]
;where, n is the line number
FONT=<font to display text in>
COLLUMN=<column on screen to start the graphic at>
ROW=<row on screen to start the graphic at>
GRASTART=<index into the font file to start the character of the graphic>
HEIGHT=<height of the graphic in terms of characters for the font file>
WIDTH=<width of the graphic in terms of characters for the font file>
```

The lines and graphic items can be in any order, and can be a maximum of 8 lines (i.e., 8 line resources and 8 graphic resources).

The default screen saver provided with VMAC consists of only four text lines, which can be customized per requirements.

The idle time out, after which the screen saver displays, is not configurable.

NOTE

- If the screen saver resource contains an invalid entry, the screen saver does not work.
- The backlight is switched off to save the power, when the terminal is running on battery and in the screen saver mode.

Setting Activation Retry Time

To modify the duration that FrontEnd retries activating the task in single application scenario, set the value, in seconds, in `#ACTIVATION_RETRY_TIMER`, the `CONFIG.SYS` variable in GID1. The default value is 5, in seconds. The maximum value is 86400 seconds.

FrontEnd retries activating the task automatically until the time out specified in `#ACTIVATION_RETRY_TIMER`. After this, FrontEnd displays the error screen and requests a key press to retry activation.

NOTE

- If the value set in `#ACTIVATION_RETRY_TIMER` is 0 or negative or invalid FrontEnd retries activating the task automatically forever.
- FrontEnd displays an error message if it tries to activate a task already exited. In a single application scenario, FrontEnd does not retry activating this task.

Enabling and Disabling Clock Display

The Clock Support is conditionally built into the FrontEnd screen by default. The display of Clock is customizable and specified in the FETIME, FEDATE config entries or file `femessag.ini`. A configurable option is given in the file `CONFIG.SYS` for enabling and disabling the Clock Support. The Clock displays only when the config variable FECLOCK is set to 1. Any value other than 1 to the config variable disables the Clock display on the FrontEnd screen. This option is given to facilitate the users to disable and enable the Clock display at runtime though the format of date and time display exists in the config variables FECLOCK, FETIME or in the `femessag.ini` file.

Configurable Application Activation Time

SWITCHTIME is a configuration variable used to set the time in ms for activating application when selected from the FrontEnd. The application should set this configuration variable in GID15. This variable is not set by default.

NOTE

As the SWITCHTIME configuration variable is in GID15, the value set by the last application overrides the previous ones.

Clock Support

The Date and Time displays on the first line of the FrontEnd screen. Date displays on the top left corner and Time displays on the top right corner of the FrontEnd screen if the config variable `FECLOCK` is set to 1. The format of the Date and Time is taken from the respective config entries `FEDATE` and `FETIME`. If these config entries are not included in the config file, then the formats are taken from the entries in the `femessag.ini` file.

The display of date and time need an optional section [`FE_DATE_TIME_STRING`] in the file `FEMESSAG.INI`. Following is an example:

```
[FE_DATE_TIME_STRING]
LINE1=MM/DD/YY
LINE2=HHH:MM
```

Date

The first entry in the `FE_DATE_TIME_STRING` is exclusively reserved for format of the date to be displayed on the FrontEnd. The valid entries in the LINE1 of the `FE_DATE_TIME_STRING` are:

MM/DD/YY (default)

DD/MM/YY

DD-MM-YY

DD/MM/YYYY

MM-DD-YY

MM-DD-YYYY

MM/DD/YYYY

YY/MM/DD

YYYY/MM/DD

YY/DD/MM

YYYY/DD/MM

YYYY-DD-MM

YYYY-MM-DD

DD-MM-YYYY

YY-MM-DD

YY-DD-MM

Time

The second entry in the `FE_DATE_TIME_STRING` is reserved for the time to be displayed on the FrontEnd. The time can be displayed in two different formats, 12-hour and 24-hour. The content valid in LINE2 are:

HH	= Hours (time displays in 12 hour format)
HHH	= Hours (time displays in 24 hour format)
MM	= Minutes
SS	= Seconds

Any other character inserted displays in the FrontEnd as it is.

Number of characters that can be displayed in VX520 terminals in a single line are:

- 8 characters for Large font type
- 16 characters for Medium font type

- 21 characters for Small font type

NOTE

- Date and Time formats are currently in upper case alphabets. All lower case formats are ignored and displayed as they are on the FrontEnd screen.
- If the Date and Time formats exceed the number of characters that can be displayed, then the characters will overlap.

If FETIME and FEDATE config variables are not included in the CONFIG.SYS and the FE_DATE_TIME_STRING section is also not included in the femessag.ini file, the default Clock is displayed depending on the FECLOCK config entry.

NOTE

Format string explanation for femessage.ini entries is also applicable for CONFIG.SYS entries.

Default date format is MM/DD/YY and time format is HHH:MM.

Date and time are displayed in the font-type (#FONTNAME) and font-size (#FONTSIZE) defined in the CONFIG.SYS. If the Clock is enabled, only three applications display in the FrontEnd screen.

Reordering Menu Items for Different Terminals

This section discusses the menu layouts for different types of terminals.

Reordering Menu Items for Verix eVo Terminals

The FrontEnd menu items may be reordered by providing the TOPMENU and BOTTOMMENU config variables. The TOPMENU variable refers to the list of applications to be listed at the first page. The BOTTOMMENU variable refers to applications to be listed at the last page.

Both config variables are comma-delimited. The names to be listed will be mapped accordingly to the Text entries of the menu in the MenuText section of the INS file. If the names cannot be mapped, it will be ignored.

For example, the following applications are downloaded in this order: EASYID, Tendercard, Gift, Check, SoftPay. If the following entries are set for TOPMENU and BOTTOMMENU config variables:

TOPMENU=SoftPay, Check

BOTTOMMENU=EasyID

Softpay and Check will be the first entries displayed in the primary FrontEnd menu. EASYID will be listed as the last entry. Tendercard and Gift will be between the Top and Bottom menu items.

NOTE

More than four entries may be listed in TOPMENU and BOTTOMMENU config variables.

TOPMENU variable will work even if BOTTOMMENU variable is not set and vice versa. If any of the other config variables are not set, the application names will be listed based on the order of download.

NOTE If any application is loaded after the TOPMENU variable and BOTTOMMENU variable are set in CONFIG.SYS, they are listed between TOPMENU or BOTTOMMENU item in the order they are loaded.

Sample Screens

The following screens show the ordering of the menu items.

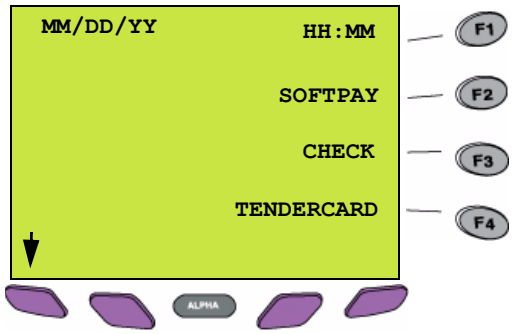


Figure 1 Primary FrontEnd Menu Sample Screen

Press the first Programmable Function (PF1) key to scroll down and view the following screen:

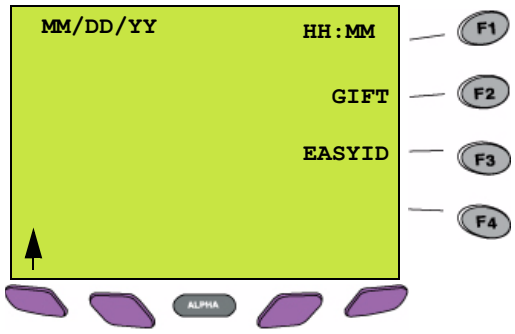


Figure 2 Primary FrontEnd Menu Sample Screen

Secondary FrontEnd Menu

The OTHERMENU config variable provides FrontEnd with a secondary VMAC menu. OTHERMENU config variable is a comma-delimited entry similar to TOPMENU and BOTTOMMENU config variables. The names to be listed will be mapped accordingly to the text entries of the menu in the MenuText section of the INS file.

If OTHERMENU config variable is set, a menu entry **OTHER** will be displayed as the last entry of the primary FrontEnd menu. If it is not set or if the entries are invalid, the menu entry **OTHER** will not be shown. If **OTHER** is selected from the primary menu, it will display the set of items listed in the said config variable.

NOTE

A maximum of 8 entries can be set to the OTHERMENU variable. If more than 8 entries are added, entries after the 8th will be displayed in the main menu.

Sample Secondary FrontEnd Screens for Verix eVo Terminals

Using the same sample screen (Figure 1 and Figure 2) and list of applications on the reordering menu Items, AutoDL is also downloaded to the terminal. Set the OTHERMENU configuration variable to AutoDL. The following screens show the ordering of the secondary menu items:

1 Pressing PF1 key in the FrontEnd Menu opens the following screen:

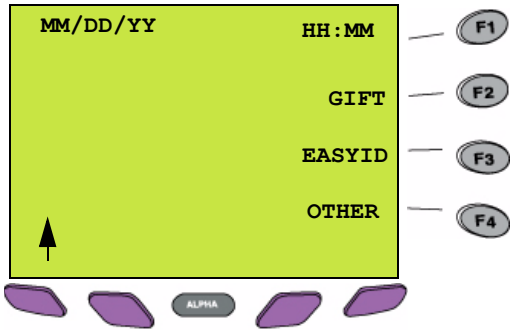


Figure 3 Primary FrontEnd Menu Sample Screen Scroll down

2 Choose **OTHER** by pressing F4 key to view the following screen:

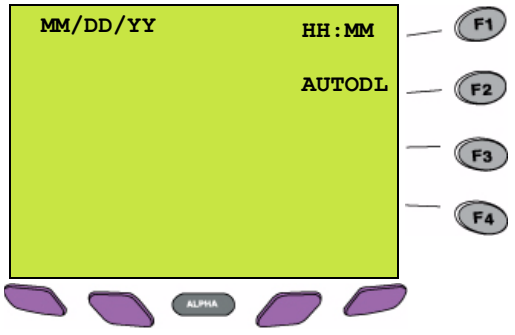


Figure 4 Secondary Menu Sample Screen

3 Press **Cancel** key to return to the **FrontEnd** Menu.

Latent Applications

Latent Applications are VMAC applications which do not have any menu item displayed. They may be activated by pressing any of the numeric keys in FrontEnd. To make an application latent, insert the section entries in the .INS file:

[ApplicationData]

LATENT=Flag

LATENTKEY=Keycode

Latent Flag

The section entry LATENT will be used for this purpose. A value 1 indicates that the application should be latent. Any other value and the application name will be shown in the menu.

Latent Keycode

Valid values for the section entry LATENTKEY are 0-9. This will be the numeric key assigned for the latent application.

NOTE

In the case when an application has more than one Activation Events, only the first entry of MenuText corresponding to the first Activation Event is hidden in the menu. Also, in the case were two or more applications having the same latent keycode, the first downloaded application will have the privilege of owning the latent keycode. Only 10 applications can be downloaded and run with latent key. All applications in excess of 10 will still be downloaded as latent, however, these will be hidden and will not be accessible.

Application Password

Each VMAC application in the VMAC menu may be password protected. For the VMAC menu password, the configuration variable format is APPNAME_PASSW. The APPNAME prefix will refer to the VMAC logical application name. Example, for the VMAC application named CHECK, the configuration variable name expected will be CHECK_PASSW. If the configuration variable is not provided, no password prompting will take place.

NOTE

Application password can be set with a combination of alphanumeric and special characters. Password length ranges from 1 to 10 characters.

To set application arguments

1. Set the runtime application arguments in the *ARG CONFIG.SYS variable.

   Assume that an application with the logical name SoftPay needs to pass r1 a2 j3 u4 as command line arguments. SoftPay must define *ARG=r1 a2 j3 u4 in the GID of the RAM where it resides.

   The first application argument starts in “argv[1];” the last argument is _FRONTENDn where, *n* is the number of applications activated, including this application. This argument helps distinguish between activation using *GO and FrontEnd activation.

To activate an application listed from FrontEnd

1. Select an application by pressing the key to the right of the required menu option.
2. To return to FrontEnd from another application, press the HOTKEY.

The HOTKEY defaults to the asterisk (*) key. You can only return to FrontEnd when an application has enabled the HOTKEY.

NOTE

- If FrontEnd remains idle for a long time, it enters a "screen saver" mode. To return to the menu from this mode, press any key.
- Refer to the Enabling/Disabling the Screen Saver section for disabling the screen saver and configuring the screen saver.

NOTE

Applications should enable switching only during idle mode and disable it as soon as the user starts an application function, feature or transaction. This prevents interruption of critical application activities if the user inadvertently presses the HOTKEY.

- The application must redraw the display and initialize any required devices on receipt of an OS activate event (`EVT_ACTIVATE`).

CAUTION

The application must close all devices on receipt of an OS deactivate event (`EVT_DEACTIVATE`).

Console ownership transfer from an application to FrontEnd is done using the Verix eVo HOTKEY mechanism.

When an application is selected using FrontEnd, an `EVT_ACTIVATE` event is posted to the application. The application receives this event and assumes ownership of the console. The following code illustrates the basic commands required for FrontEnd to receive console ownership:

```
/* Make sure the HOTKEY is enabled */
enable_hot_key();
/* Wait for an event indicating that console ownership has been
passed to the application */
while ((event & EVT_ACTIVATE) != 0L)
{
event = wait_event();
}

/* Continue with console processing until an EVT_DEACTIVATE event is
received */
```

NOTE

The above source applies when coding directly to the SDK layer.

Applications based on ACT handle `ACTIVATE_IN` and `DEACTIVATE_IN`.

Displaying Menu Text

1 The individual application must provide the text for a menu option in a text file. The naming convention is:

```
{application name}.INS
```

where, {application name} is the logical name of the application

2. The `.INS` file contains the text to display as the menu text is terminated by a new line character.
3. The `.INS` file must be loaded into the same GID as the associated application.

HOTKEY Configuration

The HOTKEY is the key used to return to Application Menu Manager from a selected application. This key is configurable. The following items apply when using HOTKEYs:

To determine the HOTKEY, FrontEnd looks for `#HOTKEY` in `CONFIG.SYS` of GID1. If this is not found, the HOTKEY defaults to the Asterisk key.

To Configure the HOTKEY

Specify the keycode of the key to be used as HOTKEY as follows:

```
#HOTKEY = \xAB
```

where, AB is the hexadecimal keycode value.

For example, `#HOTKEY=\x2a` sets the HOTKEY to the asterisk (*) key.

Dual key press also can be set for HOTKEY.

For example, to set the HOTKEY to [ENTER] + 9:

`#HOTKEY="\xe9"` in GID1.

Examples for dual key codes:

Table 7 Dual Key Codes

Dual Key	Decimal key code	Hexadecimal equivalent
[ENTER] + 5	229	0xE5
[ENTER] + 6	230	0xE6
[ENTER] + 8	232	0xE8

NOTE

If an invalid keycode value is specified for the terminal, the behavior is undefined by the operating system.

CAUTION

Do *not* configure the HOTKEY to be the same as the menu scroll key or any other menu item selection keys.

Reports

This section discusses the report types for different terminals.

NOTE

- The reports illustrated in this section are sample reports. The numbers may vary based on the application residing on the terminal.
- Printing option is available and enabled only if the terminal has a printer.

Table 8 Reports

Terminal Type	Reports
VX520	Summary Report
	Detail Report
	Device Report
VX680	Application Report
	Device Reports
	Check Files
	Summary Report
	Devman Reports
Refer to the *Verix eVo Multi-App Conductor for Touch screen Terminals Users Guide* for more details on the VX680 reports.	

FrontEnd support three types of reports for VX520 terminals. The first report is the Summary Report, a general report that shows summary information. The second report is a more detailed report that contains a list of file groups, total files, memory usage, etc. The third report shows the Device Owner Utility idle menu.

Display the report menu by pressing PF4 key on the terminal. Refer to the Figure 5 for an example terminal function key layout.

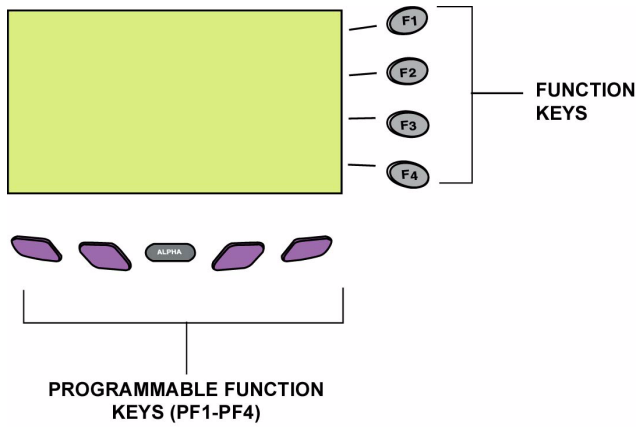


Figure 5 Terminal Key Layout

The following is an example of the report menu.

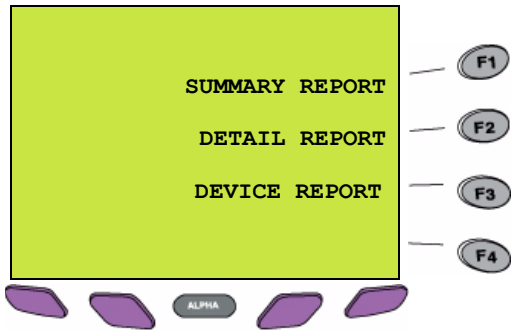


Figure 6 Report Menu Sample Screen

Press **Cancel** key to return to the FrontEnd menu that displays all the applications downloaded to the terminal.

Summary Report

Press F1 key in the report menu to generate a summary report. The summary report contains all the files registered to IMM and their GIDs, whether their status is executable or not. This report also prints all the library files in GID15.

NOTE

- The formats of the Date and Time printed on the reports are as specified in the `FEDATE` and `FETIME` config variables respectively. If these variables are not configured, the formats are taken from the section `FE_DATE_TIME_STRING` in the `femessag.ini` file. If this section is also not included in `femessag.ini` file, the date and time are printed in the default formats (MM/DD/YY for Date, and HHH:MM for Time) as explained in the Clock Support section.
- Use **Cancel** key to abort printing any report.

Detail Report

Press F2 key to display the **DETAIL REPORT** menu. Figure 7 is an example of the format of the **DETAIL REPORT** menu.

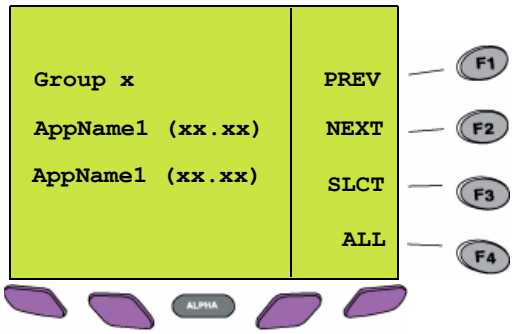


Figure 7 Detail Report Menu Sample Screen

The **DETAIL REPORT** menu displays the group number and application logical name on the left portion of the screen. The application logical name shown will be followed by the version number in parenthesis.

NOTE

- The application version number will be read from the VERSION entry under the ApplicationData section in the `.INS` file. If VERSION is blank or is not set, no version information will be displayed.
- Detail report prints the total amount of memory used in each GID.

The Detail Report screen has four options to navigate through the report menu:

- PREV (F1): Displays the GID and application logical name of the previous group. When the PREV (F1) option is selected and the GID is one, GID15 is selected and the respective application logical name displays.
- NEXT (F2): Displays the GID and application logical name of the next group. If the GID is 15 and the NEXT (F2) option is selected, GID1 is selected and the respective application logical name displays.
- SLCT (F3): Prints a detailed report of the current GID.
- ALL (F4): Prints a detailed report of all GIDs.

Device Report

Press F3 key to display the **DEVICE REPORT** menu. The following is an example of the report menu format.

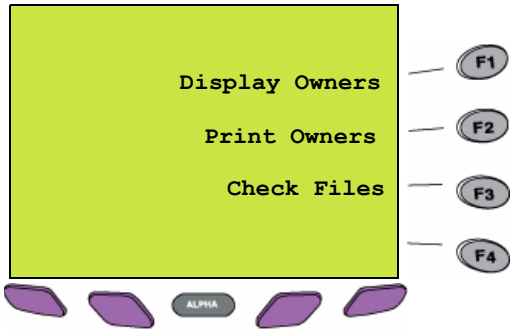


Figure 8 Device Report Menu Sample Screen

The **DEVICE REPORT** menu displays the idle menu of Device Owner Utility. Four options are available for the user to navigate through the report menu.

- Display Owners (F1): Displays the device owner/status. The following are the status for a particular device:
  - Not Supported - If a particular device is not present in the terminal.
  - Not Open - If the device is not opened by any application.
  - Application Name - Name of the application which owns a particular device.
- Print Owners (F2): Prints the information shown in Display Owner.

- Check Files (F3): Displays the available file handles.

NOTE

The F4 key is applicable for all terminals that have USB port.

Configuration Variables

VMAC uses three configuration variables while going into Power Save mode.

- ***POW** indicates the amount of time (specified in ms) that the OS will wait before attempting to set the unit into sleep mode. The timer starts when all application tasks have become idle. A value of 0 indicates that the system will never enter low-power mode. The default value is 60000 (sixty seconds), and the maximum setting allowed is 600000 (ten minutes).
- ***APOFF** is a configuration variable in GID15 for VX520, which specifies the time taken in ms for VMAC to go to freeze state. If this variable is not present in VMAC, it uses `*POW` variable. It will take 5000 ms to go to freeze state if `*POW` variable is also not present.
- ***FRZTEXT** is a configurable variable in GID1, which contains the text to be displayed by FrontEnd while in freeze state.

VMAC without any Applications

When VMAC is downloaded to the terminal [`dlvmac d r`], FrontEnd displays all the applications that co-exist with VMAC. In this scenario as no other applications are downloaded, VMAC FrontEnd displays "`No Applications Present`" message on the screen.

As Power Save mode depends only on SSAVER configuration variable, if `SSAVER` is set to 0 or `SSAVER` is deleted, then the terminal will go to the Power Save mode in `*POW` ms after displaying "`No Application Present`" message on the screen.

If the configuration variable SSAVER is set to 1, the terminal goes into the Power Save mode in `*POW` ms after the screen saver pops.

When VMAC is downloaded without screen saver [dlvmac d r -s] the terminal goes into Power Save mode in `*POW` ms which is independent of `SSAVER` or `FECLOCK` variables.

VMAC with Single Application

When a single application is downloaded with VMAC, VMAC FrontEnd activates the application on startup. If the application is Power Save enabled, terminal goes into Power Save mode in `*POW` ms after no activity is detected by the operating system. The variations in `FECLOCK` and `SSAVER` configuration values don't affect the Power Save mode. VMAC downloaded with or without screen save doesn't affect the Power Save mode as well.

NOTE

When the application is closed, the FrontEnd is activated with the power saving feature of VMAC with multiple applications enabled.

VMAC with Multiple Applications

When VMAC is downloaded to the terminal with two or more applications (which are also Power Save compliant), FrontEnd displays a menu with Clock display on the top most line followed by the logical names of the co-existing applications.

In this scenario, terminal using the Power Save options is dependent on `FECLOCK`, `SSAVER` configuration variables. The Terminal recovery from the Power Save mode is only if **Enter** key is pressed or terminal is DOCKED.

```
FECLOCK = 1/0 or deleted, SSAVER = 1
```

The time taken by the terminal to go into Power Save mode is [Time taken for the screen saver to popup] + [`*POW` ms].

```
FECLOCK = 1, SSAVER = 0 or deleted or VMAC Downloaded
withoutscreen saver [dlvmac d r -s]
```

Terminal goes to freeze state in `*APOFF` ms displaying the content of the `*FRZTEXT` instead of Clock, and then goes into the Power Save mode in next `*POW` ms. Altogether it takes [`*APOFF`+ `*POW`] ms for the terminal to go into Power Save mode. Any event in between the intermediate states, it will take [`*APOFF`+ `*POW`] ms again to go into Power Save mode.

NOTE

Any event will recover the terminal from the freeze state, to recover the terminal from Power Save mode, press **Enter** key.

```
FECLOCK = 0 or deleted, SSAVER = 0 or deleted or
VMACDownloaded without screen saver [dlvmac d r -s]
```

Terminal will go into Power Save mode in *POW ms.

Reports Menus

From **FrontEnd Reports** menu, the terminal takes `*POW` ms to go to Power Save mode.

Flexi-Records

This chapter explains the flexi-record architecture and provides a reference for developers who would like to use the flexi-record APIs. This chapter covers the following:

- Flexi-record implementation architecture
- Flexi-record API functions library

Flexi-records are used by the EESL layer as a means of formatting messages sent using the IMM. The custom event and custom event data are stored in the flexi-record fields.

Flexi-records provide a mechanism and format for storing sets of related data elements of varying types and sizes. Flexi-records also provide a mechanism to bind and store data elements not in contiguous segments of memory, while maintaining the external appearance of a single data set.

Flexi-records store transaction and configuration data within an application. In most cases, flexi-records store transient data (stored only for the duration of the transaction).

- Use flexi-records to store configuration data and read from a file at the task initiation phase and until the task halts.
- Use flexi-records as intermediate storage for state tables and other core application data.
- In some cases, flexi-records can be used as a storage template when writing data to a file.
- Use flexi-records as a mechanism to format messages.

The reasons for using flexi-records over structures to store data are as follows:

- Flexi-records escape the inflexibility of structures in C and C++, allowing addition of data elements of varying size and type.
- Flexi-records provide a single library interface for accessing all types of transaction data. This provides robustness in that the users only concern themselves with a single interface, and not with a large number of different structure definitions.
- Flexi-records provide backward compatibility between the components of an application. For example, when adding a new field to a flexi-record within a component, all components that use the flexi-record but do not need the new field, remain unchanged.

Architecture

The following is an overview of basic principals and implementation architecture behind flexi-record functionality.

Flexi-Record Structure

There are two parts to a flexi-record: First is the record header structure; Second is a variable number of field structures. Table 9 presents the header structure.

Table 9 Flexi-Record Header Structure

Element Name	Size	Element Description
Record Attributes	1 byte	Indicates the start of a flexi-record. The default byte is currently set to 17. For future releases, the value could be combined with other attributes to indicate different versions of flexi-records or encryption mechanisms used.
Filler	1 byte	Compensates for the even byte alignment requirement on the Omni 2000 platforms.
Field ID Maximum	2 bytes	Specifies the maximum Field ID value used when storing or retrieving fields. This means that users can be restricted to a specific range of Field IDs. The default Field ID maximum is 65535.
Field ID Minimum	2 bytes	Specifies the minimum Field ID value used when storing or retrieving fields. This means that users can be restricted to a specific range of Field IDs. The default Field ID minimum is 0.
Record Size	2 bytes	Specifies the size of the flexi-records, in bytes. This includes the sum of the sizes of the record header, the field headers, and each individual field.
Maximum Record Size	2 bytes	Specifies the maximum record size allowed. This is limited by the size of the data buffer provided when initializing the flexi-records.

Flexi-Record Fields

Table 10 lists flexi-record field structures.

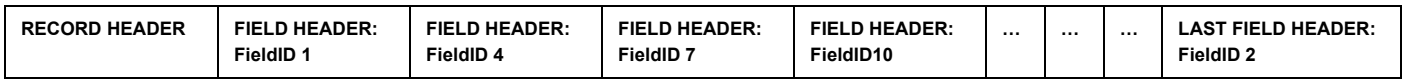
Table 10 Flexi-Record Field Structures

Element Name	Size	Element Description
Field Length	2 bytes	Length of the field data following the field header.
Field ID	2 bytes	Field ID that uniquely identifies the field within the flexi-record.
Field Data Type	5 bits	Specifies the field data type. The default is always 00000, which indicates that the field contains raw byte data.
Field Attributes	3 bits	Field attributes describing field access rights. The default is always set to 000, indicating there are no restrictions associated with the field.

Table 10 Flexi-Record Field Structures (continued)

Element Name	Size	Element Description
Filler	1 byte	Compensates for the even byte alignment requirement on the Omni 2000 platform.
Field Data	Field length in bytes	Binary user data associated with the field.

To create a flexi-record, apply these structures to a buffer provided by the flexi-record user. The following illustrates the flexi-record structure:



Flexi-Record Field ID

By specifying a unique Field ID, any field within a flexi-record is accessible. A new field can be added to a flexi-record by specifying a unique Field ID associated with the new field, as well as the field data and length of the field data (refer to the Flexi-Record Functions section for more details).

Usage and Maintenance

1. Declare all Field IDs as constants in a standard include file for C or C++.

   For any base application, declare a set of basic Field IDs (refer to the Automatic Application Selection for a list of example Field IDs).

2. Define the base set of Field IDs for a generic base application that can be customized for individual markets.

   For each customization, local developers can add new Field IDs as required to complete the application.

Flexi-Record Functions

The following functions are available in the flexi-record module, `VARREC`.

To use flexi-record functionality; link the `eesl.a` library with the application object file. The function definitions, as well as supported attributes, are specified in the `VARREC.H` include file.

The following defines the variable record library's set of error values:

```
#define VAR_NO_ERROR                          0
#define VAR_INSUFFICIENT_BUFFER_SPACE         1
#define VAR_FIELD_NOT_FOUND                   2
#define VAR_BUFFER_UNINTIALIZED               3
#define VAR_NULL_POINTER                      4
#define VAR_FIELD_PROTECTED                   5
#define VAR_INVALID_FIELD_ID_SUBSET           6
#define VAR_INVALID_DATA_TYPE                 7
#define VAR_CALLER_BUFFER_SIZE_INSUFICCIENT   8
#define VAR_SECONDARY_BUFFER_UNINITIALIZED    9
#define VAR_INVALID_ATTRIBUTE                 10
```

The following are the APIs provided by VMAC.

vVarInitRecord()

Initializes the buffer provided to store a new flexi-record structure. This is a flexi-record public interface function.

Prototype

```
void vVarInitRecord(unsigned char * recordBuffer, short bufferLen,
                    unsigned char attributes);
```

Parameters

`recordBuffer`	Flexi-record buffer allocated by the caller.
`bufferLen`	The size of the buffer provided. The total flexi-record size is not allowed to exceed the specified length.
`attributes`	Attributes to identify the type of flexi-record for example, identify the encryption mechanism used. RFU (Reserved for Future Use).

Return Values

No return value.

shVarAddField()

Appends a new field to an existing flexi-record. This is a flexi-record public interface function.

Prototype

```
unsigned short shVarAddField(unsigned char * recordBuffer,unsigned short
                  fieldId,unsigned char * fieldData,unsigned short
                  dataLength);
```

Return Values

Returns a non-zero value on error.

Parameters

`recordBuffer`	Initialized flexi-record buffer allocated by the caller.
`fieldId`	The field ID of the new field being added to the flexi-record.
`fieldData`	Buffer containing the data to insert into the new field.
`dataLength`	Size of the data to insert into the new field.

shVarGetField()

Retrieves a specified field from within the flexi-record. This is a flexi-record public interface function.

Prototype

```
unsigned short shVarGetField(unsigned char * recordBuffer,unsigned short
                             fieldId,unsigned char * fieldData,unsigned
                             short   fieldBufferLength,unsigned short  *
                             fieldDataLength);
```

Parameters

`recordBuffer`	Initialized flexi-record buffer.
`fieldId`	The field ID of the field being retrieved.
`fieldData`	Caller-allocated buffer where the field data is copied.
`fieldBufferLength`	Size of the caller-allocated buffer.
`fieldDataLength`	Actual size of the field data within the flexi-record.

Return Values

Returns a non-zero value if an error occurs locating the field.

shGetRecordLength()

Returns the length of the flexi-record. This is a flexi-Record public interface function.

Prototype

```
short shGetRecordLength(unsigned char * pucRecordBuffer, short  *
                                    inRecordLength);
```

Parameters

`recordBuffer`	Initialized flexi-record buffer.
`inRecordLength`	Assigned current length of the flexi-record, including the combined length of the Field Headers and the length of the Record Header.

Return Values

Returns a non-zero value on error.

shVarDeleteField()

Deletes a field from a flexi-record. This is a flexi-record public interface function.

Prototype

```
unsigned short shVarDeleteField(unsigned char * recordBuffer,short
                                inFieldId);
```

Parameters

`recordBuffer`	Initialized flexi-record buffer.
`fieldId`	The field ID of the field to delete.

Return Values

Returns a non-zero value on error.

shVarQueryField()

Returns the field attributes.

Prototype

```
unsigned short shVarQueryField(unsigned char * recordBuffer,unsigned short
                               fieldId,unsigned short  *
                               fieldDataLength,unsigned char *
                               type,unsigned char * attributes);
```

Parameters

`recordBuffer`	Initialized flexi-record buffer.
`fieldId`	Field ID of the field being queried.
`fieldData`	Caller-allocated buffer where field data is copied to.
`dataLength`	Size of the caller-allocated buffer.
`fieldDataLength`	Actual size of the field data within the flexi-record.

Return Values

Returns a non-zero value if an error occurs locating the field.

shVarUpdateField()

Updates the data in a flexi-record field. This is a flexi-record public interface function.

Prototype

```
unsigned short shVarUpdateField(unsigned char * recordBuffer,unsigned
                                short  fieldId,unsigned char *
                                fieldData,unsigned short  dataLength);
```

Parameters

`recordBuffer`	Initialized flexi-record buffer.
`fieldId`	Field ID of the field being updated.
`fieldData`	Buffer containing the new data to replace the existing Field Data.
`dataLength`	Size of the new Field Data.

Return Values

Returns a non-zero value on error.

Extended Flexi-Record API

ushInitStandardFlexi()

Assigns the buffer passed as a parameter to the global space of the Extended Flexi-record API. This is a mandatory function call for the rest of the Extended Flexi-record APIs. The buffer should be initialized using vVarInitRecord() function before calling this API.

Prototype

```
unsigned short ushInitStandardFlexi(unsigned char * recordBuffer);
```

Parameters

`recordBuffer` Initialized flexi-record buffer.

Return Values

Return non-zero value on error.

Example

```
unsigned char queueData[1024] = {0};
VarInitRecord (queueData, sizeof(queueData), 0);
ushInitStandardFlexi(queueData);
```

NOTE

It is mandatory to initialize the buffer that is used to add or delete flexi-record field(s) before using any of the extended flexi-record APIs except `ushInitStandardFlexi()` API.

shVarAddUnsignedInt()

Adds an unsigned int to the Flexi-record.

Prototype

```
unsigned short shVarAddUnsignedInt(unsigned short fieldId,
                                   unsigned short fieldData);
```

Parameters

`fieldId`	Field Identifier of the field being added to the flexi-record.
`fieldData`	Contains the data to be added to the field ID.

Return Values

Return non-zero value on error.

Example

```
#define FIELD_ID_INT 100
unsigned int FieldData = 0;
shVarGetUnsignedInt(FIELD_ID_INT, FieldData);
```

shVarGetUnsignedInt()

Gets the unsigned int from the Flexi-Record.

Prototype

```
unsigned short shVarGetUnsignedInt(unsigned short fieldId,
                unsigned short * fieldData);
```

Parameters

`fieldId`	Field Identifier of the field being added to the flexi-record.
`fieldData`	Parameter in which the data is to be filled.

Return Values

Return non-zero value on error.

Example

```
#define FIELD_ID_INT 100
#ifdef _TARG_68000 //Verix
    unsigned int    fieldData;
#elif __thumb   //Verix V
    unsigned short  fieldData;
#endif //_TARG_68000
shVarGetUnsignedInt(FIELD_ID_INT, &fieldData);
```

shVarGetUnsignedChar()

Gets the unsigned char from the flexi-record.

Prototype

```
unsigned short shVarGetUnsignedChar(unsigned short fieldId,
                unsigned char * fieldData);
```

Parameters

`fieldId`	Field Identifier of the field being added to the flexi-record.
`fieldData`	Parameter in which the data is to be filled.

Return Values

Return non-zero value on error.

Example

```
#define FIELD_ID_CHAR 101
unsigned char    fieldData;
shVarGetUnsignedChar (FIELD_ID_CHAR, &fieldData);
```

shVarAddUnsignedChar()

Adds the unsigned char from the flexi-record.

Prototype

```
unsigned short shVarAddUnsignedChar(unsigned short fieldId,
                 unsigned char fieldData);
```

Parameters

`fieldId`	Field Identifier of the field being added to the flexi-record.
`fieldData`	Contains the data to be added to the field ID.

Return Values

Return non-zero value on error.

Example

```
#define FIELD_ID_CHAR 101
unsigned char FieldData = 0;
shVarAddUnsignedChar (FIELD_ID_CHAR, FieldData);
```

shVarAddUnsignedLong()

Adds the unsigned long from the flexi-record.

Prototype

```
unsigned short shVarAddUnsignedLong(unsigned short fieldId,
                                    unsigned long fieldData);
```

Parameters

`fieldId`	Field Identifier of the field being added to the flexi-record.
`fieldData`	Contains the data to be added to the field ID.

Return Values

Return non-zero value on error.

Example

```
#define FIELD_ID_LONG 102
unsigned long FieldData = 0;
shVarAddUnsignedLong (FIELD_ID_LONG, FieldData);
```

shVarGetUnsignedLong()

Gets the unsigned long from the flexi-record.

Prototype

```
unsigned short shVarGetUnsignedLong(unsigned short fieldId,
                    unsigned long * fieldData);
```

Parameters

`fieldId`	Field Identifier of the field being added to the flexi-record.
`fieldData`	Parameter in which the data is to be filled.

Return Values

Return non-zero value on error.

Example

```
#define FIELD_ID_LONG 102
unsigned long FieldData = 0;
shVarGetUnsignedLong (FIELD_ID_LONG,&FieldData);
```

LOGSYS Debug Utility

This chapter explains how to debug applications written to run in the Verix eVo Multi-Application Conductor environment. Applications that communicate with other applications (such as, custom events using EESL) in this environment are difficult to debug. Some factors that contribute to multi-application debug complications are:

- When using the debug utility to debug applications, error conditions can occur. This is because other applications running on the terminal may rely on an *instant* response from the application being debugged (for example, a failure to respond to CLASS – A custom events within 350 ms results in an error when using EESL_send_event() function).
- When using the debug utility, the application is often in the idle state (i.e., no processing is occurring). This means that synchronization cannot occur between the application being debugged and other applications running on the terminal.
- When running the debug utility on a set of applications, all applications in the set are in the idle state while the debug continues on a single application within the set. This means that all applications within the set, except the one in the debug utility process, will not respond to any OS or custom events.

The LOGSYS Library Utility

To assist in the debug process, VMAC provides a set of macros that can be used to write the debug statements to a file in a terminal RAM and/or sent to the terminal serial communications port (COM1 or COM2 or COM6 or USBD). These macros are provided in the LOGSYS.H file.

The typical LOGSYS configuration has LOGSYS initialized to send a debug statement to the Verix eVo COM1 or COM2 or COM6 or USBD port. The port to use for logging messages is set by the #LOGPORT CONFIG.SYS variable. This variable is set in GID15.

To enable logging on COM port 1 of the Verix eVo terminal set #LOGPORT=COM1, for COM port 2 #LOGPORT=COM2, for COM port 6 #LOGPORT=COM6, and for USBD port #LOGPORT=USBD. If the #LOGPORT environment variable is not found in GID15, logging messages are disabled.

The statements can be viewed using a PC-based terminal emulator set to the following communications parameters:

- 115200 baud
- 8N1 (8 bits, no parity, 1 stop bit)

It is possible for multiple applications to simultaneously send LOGSYS statements to the COM ports. This is because each call to a LOGSYS macro within an application opens the COM ports. If the COM port is not available, the LOGSYS macro waits until the port is available. The COM port closes once the macro process completes.

LOGSYS File Size

The size of LOGSYS (`VMAC.LOG`) file can be set in GID15 using `*VMACLOGSYSMAX` configuration variable.

For Verix eVo

The maximum size of the LOGSYS file can be set up to 102400 bytes. By default the value is taken as 102400 bytes if the `*VMACLOGSYSMAX` is not set.

The `VMAC.LOG` file is created at the run time. The size of the file increments as and when the action is logged into the LOGSYS file. Once the size reaches its maximum limit, the logging will start from the beginning of the file. Where the existing logs are replaced with the new logs.

LOGSYS Command Filters

It takes 10 ms to complete each call to a LOGSYS macro. If an application is making multiple LOGSYS calls (more than 10 per second), application performance degrades. If called frequently, LOGSYS calls can also alter synchronization between applications. This is because an application making numerous LOGSYS calls may fail to process all incoming `CLASS - A` custom events. For this reason, there must be a provision to turn off individual logging statements or sets of logging statements. LOGSYS filters provide a way to turn off logging statements.

A single bit within a `long` value represents a LOGSYS filter. There are 32 possible filter values. The filter value is passed as a parameter in each LOGSYS API call. LOGSYS API function calls can then be turned on or off when the LOGSYS library is initialized by specifying which filters to use. For example, the following filter is defined:

```
#define EESL_DEBUG_SEND_SUCCESS 0x00000080L
```

A call is made within an application to the following LOGSYS macro:

```
LOG_PRINTFF(EESL_DEBUG_SEND_SUCCESS, "This error occurred: %d",error);
```

The following call to the LOGSYS initialization function activates the statement, allowing the debug text to be output to COM1.

```
LOG_INIT(APP_LOGICAL_NAME, LOGSYS_COMM, EESL_DEBUG_SEND_SUCCESS);
```

The error logging macros are:

```
LOG_NONZERO_ERROR(a,b)
LOG_NEGATIVE_ERROR(a,b)
LOG_ZERO_ERROR(a,b)
LOG_NONZERO_ERROR_CRIT(a,b)
LOG_NEGATIVE_ERROR_CRIT(a,b)
LOG_ZERO_ERROR_CRIT(a,b)
```

```
LOG_NULL_POINTER(a,b)
```

The message logging macros are:

```
LOG_PRINTF(a)
LOG_PRINTFF(a)
```

LOGSYS Output

The output can be to COM1/COM2/COM6/USBD port or to a file, or to both. The output of the LOG_INIT() or LOG_PRINTFF(()) macro is as follows:

```
{Logical App name}: F: {file name} | L: {Line number}| {Text provided by
caller}
```

The output of the error logging macros is as follows:

```
{Logical App name}: ER {Error value}| F: {file name} | L: {Line number}
```

NOTE

LOG_PRINTF(()) family functions can handle maximum message length of 1100 bytes, including the header.

Activate the LOGSYS Debug Utility using CONFIG.SYS Variables

This section explains how to override existing LOGSYS initialization parameters (refer to the LOG_INIT() section for more details) by setting the following specific CONFIG.SYS parameters.

Override the Debug Output Destination

To override the default output destination, set the following CONFIG.SYS variable in the GID where the application resides:

```
{Application Logical name}LOG
```

For an application with the logical name TMAIL, set the CONFIG.SYS variable TMAILLOG. The variable is set to one of the following output media:

- C — Output to the COM port (COM1 or COM2 or COM6 or USBD).
- F — Output to a file.
- B — Output both to the COM port (COM1 or COM2 or COM6 or USBD) and a file.
- N — Disables the debug output.

Enable Application Filters

To enable application filters, set a CONFIG.SYS variable in the same GID as the target application. The CONFIG.SYS variable has the following format:

```
{Application Logical name}FIL
```

The CONFIG.SYS variable is set to a C format hex string, as follows:

```
TMAILFIL=0x00000008
```

The hex value corresponds to the LOGSYS filter to activate.

Compile Time Option to Activate/ Deactivate LOGSYS Debugs

It is possible to activate and deactivate LOGSYS debugs by setting the compile-time switches. To activate LOGSYS statements, define the compile-time switch LOGSYS_FLAG in the make file of the application. The following example code activates LOGSYS:

```
COptions = -D__K2__ -DLOGSYS_FLAG -D_DEBUG -f -s int=2 -s sizeof=2 -O d -
r a4 -r a5 -o $(@)
```

NOTE

- An undefined LOGSYS_FLAG at compile time disables *all* LOGSYS statements.
- Setting `{Application Logical name}LOG=C` does not activate the debug statements if the `LOGSYS_FLAG` is undefined at compile time.

The LOGSYS Macros

This section presents the available LOGSYS macros.

LOG_INIT()

Initializes the LOGSYS library.

Prototype

```
void LOG_INIT(char * appName,short  logType,unsigned long filter);
```

Parameters

`appName` Logical application name of the task where the inquiry pointed.

`logType` LOG output media to use. Valid values are:

- `LOGSYS_NONE`
- `LOGSYS_FILE`
- `LOGSYS_COMM`
- `LOGSYS_COMM_FILE`

`filter` LOGSYS filter.

LOG_PRINTF(())

Writes the debug statements to COM1 or to a specified file. `LOG_PRINTF(())` function has the same parameter format as the C function `printf`.

Prototype

```
void LOG_PRINTF((char * format,...));
```

NOTE

Two brackets *must* enclose the macro arguments.

LOG_PRINTFF(())

Writes the debug statements to COM1 or to a specified file. `LOG_PRINTFF(())` function has the same parameter format as the C function `printf`, except for the first parameter, the LOGSYS filter.

The message prints only if the filter is allowed by the filter values specified in `LOG_INIT`.

Prototype `void LOG_PRINTFF((unsigned long filter, char * format,...));`

NOTE

Two brackets *must* enclose the macro arguments.

LOG_NONZERO_ERROR()

Writes the line number where a non-zero error occurred if the specified filter is allowed.

Prototype `void LOG_NONZERO_ERROR(value, filter);`

LOG_ZERO_ERROR()

Writes the line number where a zero error response occurred (i.e., zero bytes returned from a file or COM read) if the specified filter is allowed.

Prototype `void LOG_ZERO_ERROR(value, filter);`

LOG_NEGATIVE_ERROR()

Writes the line number where a negative error response occurred if the specified filter is allowed.

Prototype `void LOG_NEGATIVE_ERROR(value, filter);`

LOG_NONZERO_ERROR_CRIT()

Writes the line number where a non-zero critical error occurred if the specified filter is allowed.

Prototype `void LOG_NONZERO_ERROR_CRIT(value, filter);`

LOG_NEGATIVE_ERROR_CRIT()

Writes the line number where a critical negative error response occurred if the specified filter is allowed.

Prototype `void LOG_NEGATIVE_ERROR_CRIT(value, filter);`

LOG_ZERO_ERROR_CRIT()

Writes the line number where a critical zero error response occurred if the specified filter is allowed.

Prototype `void LOG_ZERO_ERROR_CRIT(value, filter);`

LOG_NULL_POINTER()

Writes the line number where null pointer error occurred if the specified filter is allowed.

Prototype `void LOG_NULL_POINTER(value, filter);`

Debugging

The following section provides debugging guidelines:

To debug an application

1. Download the application to debug to GID1 in RAM.
2. Download all components of VMAC and all other applications to RAM.
3. Set the following in the `CONFIG.SYS` file in GID15:

   ```
   MASTER = PRINTSP
   #DEBUGPORT = COM1 or COM2 or COM6
   ```

   `#DEBUGPORT =` sets the COM port used for debugging.
4. Download the `Dbmon.out` and `dbmon.out.p7s` files to GID1.
5. Set `*GO=dbmon.out`.

   After completing Verix eVo debug procedures, use the SDS debugger to start debugging the required application.
6. Step over the EESL_Initialise() call.

   EESL_Initialise() function calls IMM and IMM continues to start the remaining applications listed in the `default.INI` file.

LogServer

The LogServer is a service task of VMAC that is used for logging debugging statements efficiently. It is an executable that runs in GID1 and not listed in the FrontEnd. LogServer enables to collect the logs piped from the applications and directs it to the selected device.

Features

- LogServer enables to minimize the possibility of loosing the log from the applications.
- LogServer supports both Server (S) and Backward compatibility (B) modes.

Configuration

The `CONFIG.SYS` variable needs to be configured to capture logs.

Following are the `CONFIG.SYS` variables:

- `#LOGPORT` is a variable that is configured in GID15, which specifies the device to log the debugging statements. It is used to configure the COM port value to direct the logs. Refer to the The LOGSYS Library Utility section for more details.
- `<Application Logical Name>LOG` is a variable that is configured by the application. Refer to the Activate the LOGSYS Debug Utility using CONFIG.SYS Variables section for more details.
- `#LOGMODE` is a variable that is configured in GID15. The following are the available modes:
  - S mode - set the value as ‘S’ for LogServer mode. The logs are captured even if the value of <Application Logical name>LOG variable is set to F or B or P.

- O mode - set the value as ‘O’ for LogServer mode. The logs are captured even if the value of <Application Logical name>LOG variable is set to P and *debug is set to 1 in GID1.
- Backward compatibility - set the value as ‘B’ for backward compatibility mode. In this mode the logs are captured only if the value of <Application Logical name>LOG variable is set to C or B or F or P.

NOTE

- Configuration variable `#LOGMODE` should be set in GID 15.
- If `#LOGMODE` =’S’, the logs are not captured if the device is unplugged and plugged again. The terminal needs to be re-started again to get the logs.
- If `#LOGMODE` is misconfigured or not present or `#LOGMODE` = ‘B’ then the LogServer is in backward compatibility mode.
- The recommended configurations are:`<Application Logical name>LOG` =”P” when `#LOGMODE` =”S”.
- To get both OS logs and application logs, set `<Application Logical name>LOG`=”P” and `#LOGMODE` =”O” and set *debug =1 in GID 1.

Table 11 describes the behavior when LogServer or Backward compatibility mode is selected.

Table 11 Mode Behavior

LogServer	<Application> Log	Behavior
S	C	Logs are not captured.
S	F	Application logs are directed to the file.
S	B	Application logs are directed to both the COM port and a File. As this is in the Server mode logs directed to COM port are not captured.
S	P	Application logs are directed to the pipe.
B	C	Application logs are directed to the COM port.
B	F	Application logs are captured to the file.
B	B	Application logs are directed to both the COM port and a file.
B	P	Application logs are captured to the pipe which is connected to the LogServer.
O	P	To capture both application logs and OS logs.

CAUTION

It is recommended to set `<APP_LOGICAL_NAME>LOG` config variable to **P** in the application, when LOGSVR component is downloaded. It will lead to timing issues if the variable is set to **C**.

Dynamic Menu

This chapter provides references to Dynamic Menu APIs. The Dynamic Menu is used by applications to add and delete menu items at runtime.

There are two APIs available:

- `Add_Menu_Item()`
- `Remove_Menu_Item()`

The call to `Add_Menu_Item()` is not persistent. The application will need to call this API for every menu item each time the terminal is rebooted. This will avoid issues such as an application being deleted from system mode.

When `Add_Menu_Item()` is called, an event is sent to FrontEnd to add a menu item followed by an event to DEVMAN to add an entry in MDT.

`Add_Menu_Item()` then sends the received Flexi-Record to DEVMAN and FrontEnd (depending on the mode of VMAC downloaded) by adding one more field, in this case, `menu_ID` to the record. The decision whether to send the event to FrontEnd is made by querying IMM and using the `EESL_check_app_present()` API.

The `Remove_Menu_Item()` API will ensure that the menu item is removed immediately by sending appropriate messages to the FrontEnd application. Before the application removes an added menu, the application will close all open devices. The vmac.lib file will take care of sending `RES_COMPLETE_EVENT` to the DEVMAN application so that it can acquire the devices allotted to each application when the Dynamic Menu is activated.

When `Remove_Menu_Item()` is called while VMAC is running in the FrontEnd mode, the FrontEnd will remove the dynamic menu text from the FrontEnd menu. This results in the removal of the corresponding dynamic menu from MDT by DEVMAN.

Function definitions and supported attributes are specified in the included `DynamicMenu.h` file.

The following are the events defined for adding and removing dynamic menu:

- #define VMS_ADD_MENU_MDT 20499
- #define VMS_REM_MENU_MDT 20500

The following are MACRO definitions used when adding text to the Dynamic Menu:

Table 12 MACRO Definitions

#define DYN_MENU_TEXT	1
#define DYN_PRIORITY	2
#define DYN_INPUT_EVENT	3
#define DYN_OUTPUT_EVENT	4
#define DYN_DEVICE_LIST	5
#define DYN_DEVICE_COUNT	6
#define DYN_MENU_ORDER	7
#define DYN_PASSWORD	8
#define DYN_LT_KEYCODE	9
#defineDYN_MENU_ID	10
#define APP_TASK_ID	11
#define APP_GID	12

Using the Dynamic Menu Library

The Dynamic Menu include file, `DynamicMenu.h`, must be inserted directly after standard C includes and Verix eVo OS includes.

For example, the following shows the position of `DynamicMenu.h` after the C includes and Verix eVo OS include:

```
SVC.H:

#include <stdio.h>
#include <string.h>
#include <svc.h>

/* Dynamic Menu include must appear here */
/*****************************************/
#include "DynamicMenu.h"
/* Other application includes start here */
```

The following are the APIs provided by VMAC.

Add_Menu_Item()

This API creates a menu at runtime.

Prototype

```
short Add_Menu_Item(unsigned short menu_id, unsigned char * MenuInfo);
```

Parameters

menu_id	The unique menu id that the application chooses.	
MenuInfo	Menu Text	Text that will be displayed by FrontEnd.
	Priority	Priority of devices to be acquired.
	Input Event	Event to be used by FrontEnd to activate through DEVMAN.
	Output Event	Event used to notify the application which menu item is selected by the user.
	Menu Order	flag to decide on the order to add this application in FrontEnd. Accepted values are: • T - TOPMENU Item • O - OTHERMENU Item • B - BOTTOMMENU Item If this field is not present or invalid value, the default is ‘B’.
	Password	Select this application if needed.
	Device List	Required to invoke the Dynamic Menu.
	Device Count	Specifies the number of devices included in the Flexi record for the Dynamic Menu.
	Latent app key	An option when used will not display the menu by FrontEnd and will be treated as a latent menu. When the ‘latent app key’ is pressed, the output event will be sent to the application.

Return Values

Success: 0

Failure: Returns one of following non-zero values on error:

Value	Description
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this returns the same error.
EESL_FAILURE	The queue is empty.
EESL_INVALID_PARAMETER	An invalid parameter was passed to the function check for null pointers.
VAR_INVALID_DATA_TYPE	An invalid data type was passed to the function.
INVALID_MENU_ID	If the menu ID is equal to zero.

Remove_Menu_Item()

This API is used to remove previously added menu item at runtime.

Prototype

```
short Remove_Menu_Item(unsigned short menu_id);
```

Parameter

menu_id	The unique id defined by the application

Return Values

Success: 0

Failure: Returns one of following non-zero values on error:

Value	Description
EESL_APP_NOT_FOUND	The destination application specified has not registered.
EESL_ERROR_INIT_FAILURE	VMAC is not present. All calls to EESL API functions following this returns the same error.
EESL_FAILURE	The queue is empty.
EESL_INVALID_PARAMETER	Invalid parameter was passed to the function check for null pointers.
MENU_NOT_FOUND	The menu to delete is not found.
INVALID_MENU_ID	If the menu ID is equal to zero.

NOTE

VMAC will use task ID and menu_id to distinguish menu IDs from one task to another.

Device Manager Support for Dynamic Menu

This section describes Device Manager support while dynamically adding or removing a menu item:

- On receipt of the `VMS_ADD_MENU_MDT` event, the DEVMAN extracts the Flexi-Record and passes the device mask to the MDT class in its required format, to add the record to MDT.
- The task ID stored in the MDT belongs to the application which adds the dynamic menu. Based on this, DEVMAN sends the response to the input event from the dynamic menu to the application which added the dynamic menu.

NOTE

Menu IDs are unique. Applications cannot generate the same menu ID for different dynamic menus.

The `Add_Menu_Item()` API returns **SUCCESS** even if DEVMAN has already reached he maximum number of entries in MDT.

An application which added the dynamic menu should close the acquired devices before removing the dynamic menu.

FontEnd Support for Dynamic Menu

This section explains the support of FrontEnd while dynamically adding or removing a menu item.

- Dynamic menu text added by applications is added to the end of the menu, after all the application names from **.ins** files have been added.
- Maximum number of text displayed in the menu:
  - Small font 21 characters
  - Medium font 16 characters
  - Large font 8 characters
- If the application names have more characters for a particular font type, then the left side of the names will be truncated by FrontEnd while displaying.
- FrontEnd will not accept any event from applications to add and display menu text dynamically until **.ini** files are processed.
- The FrontEnd menu is updated dynamically, if:
  - Only two applications are present in the FrontEnd menu. New menu text generated is added as third application and the screen is refreshed.
  - If an application adds another menu to the FrontEnd that is already displaying the maximum (for the current screen) number of applications, then an arrow is displayed in the menu and allows the user to scroll.
  - If an arrow key is already displayed on the FrontEnd, then adding more applications will not have any effect on the screen.
- Frontline displays the new menu text in the same font that is used to display the other applications.

- The `Add_Menu_Item()` API returns **SUCCESS** even if FrontEnd has already reached the maximum number of applications.
- FrontEnd supports **TOPMENU**, **BOTTOMMENU** and **OTHERMENU** options while adding dynamic menu items.
- If the Flexi-Record sent by application through `Add_Menu_Item()` has **DYN_PASSWORD** field set, then the particular application is password protected.
- If the Flexi-Record sent by application through `Add_Menu_Item()` has **DYN_LT_KEYCODE** field set, then the particular menu is considered as a latent application.
- Menu items that are added or removed are not reflected in the summary report.
- Detailed report prints out the name of the script files present in GID.
- The flags in **MENU_ORDER** field are as follows:

T	**TOPMENU**	Refers to the list of applications to be listed in the first page.
B	**BOTTOMMENU**	Refers to applications to be listed in the last page.
O	**OTHERMENU**	The OTHERMENU provides FrontEnd with a secondary VMAC menu. OTHERMENU is a comma-delimited entry similar to TOPMENU and BOTTOMMENU. The names to be listed are mapped according to the text entries of the menu in the menu text section.

XML Parser

VMAC library supports XML Parser, which is used by the applications. This chapter provides details of the XML Parser APIs exposed by the library.

ParseXML()

This API initializes the XML Parser to parse the elements from the XML string provided into a internal data structure, which will be referred for any other operation on the XML. This API returns a root element which needs to be provided to all the XML Parser APIs.

Prototype

```
XMLElement * ParseXML(const char * lpszXML, XMLResults *pResults);
```

Parameters

`lpszXML`	String containing the XML data that needs to be parserd.
`pResults`	This is the output parameter containing the results. Refer to the below mentioned data structure and the return values for more details.

Return Value

Returns a NULL pointer but the `XMLResults` structure contains the following error codes:

`eXMLErrorNone`	Successfully parsed the XML string.
`eXMLErrorMissingTagName`	Elements in the XML does not contain proper start tag “<”.
`eXMLErrorMissingEndTagName`	Elements in the XML does not contain proper end tag “>”.
`eXMLErrorNoMatchingQuote`	String enclosed in a quote does not contain a matching end quote.
`eXMLErrorUnmatchedEndTag`	Elements in the XML are not properly tagged.
`eXMLErrorUnexpectedToken`	A close tag is found at an unexpected place.

ReadXMLFile()

This API reads an XML file and returns the content of the XML file into a buffer.

Prototype

```
Int ReadXMLFile (char *pFileName, char *pFileBuffer, int iBuffLen);
```

Parameters

`pFileName`	Fully qualified file path on the terminal.
`pFileBuffer`	Buffer provided by the application which will contain the buffer read from the file.
`iBuffLen`	Size of the buffer provided by the application.

Return Values

Success:	Returns the number of bytes read from the file.
Failure:	-1

GetXMLElement()

This API returns the value of XML element specified by the application through a fully qualified XML path.

Prototype

```
int GetXMLElement(XMLElement *pRoot, char *pPath, char *pValueBuff, int iBuffLen);
```

Parameters

`pRoot`	Root node structure returned by the ParseXML() API.
`pPath`	Full path of the XML node whose value has to be read.
`pValueBuff`	Buffer containing the value of the XML element returned by the API.
`iBuffLen`	Size of the value buffer.

Return Values

Success:	0
Failure:	-1

GetXMLAttribute()

This API returns the value of XML attribute specified by the application through a fully qualified path.

Prototype

```
int GetXMLAttribute(XMLElement *pRoot, char *pPath, const
char * pAttribName, char * pAttribValue, int iBuffLen);
```

Parameters

`pRoot`	Root node structure returned by the ParseXML() API
`pPath`	Full path of the XML node which holds the attribute to be retrieved.
`pAttribname`	Buffer containing the attribute name whose value has to be retrieved.
`pAttribValue`	Buffer containing the value of the XML attribute returned by the API.
`iBuffLen`	Size of the value buffer.

Return Values

Success:	0
Failure:	-1

DeleteXMLElement()

This API deletes the XML element specified by the application.

Prototype

```
int DeleteXMLElement(XMLElement *pEntry);
```

Parameters

`pEntry`	XML element in the structure that needs to be deleted. This element is got by using FindXMLElement() API

Return Values

Success:	0
Failure:	-1

FindXMLElement()

This API returns an XML element pointer of an element specified by the application through a fully qualified path.

Prototype

```
XMLElement * FindXMLElement(XMLElement *pEntry, const char *
lpszPath);
```

Parameters

pEntry	Root node structure returned by ParseXML() API
lpszPath	Fully qualified path of the XML element to be returned.

Return Values

Success:	Returns a valid pointer to a XML element.
Failure:	-1

DeleteXMLRoot()

This API is similar to the DeleteXMLElement() API except that this API also clears the root node provided as part of this API.

CreateXMLStringR()

This API creates a user friendly XML string from a given element with appropriate white space and carriage returns.

Prototype

```
int CreateXMLStringR(XMLElement * pEntry, LPTSTR lpszMarker,
int nFormat);
```

Parameters

pEntry	Root node structure returned by ParseXML() API.
lpszMarker	String to create results. This is zero while calculating the size of the returned string.
`nFormat`	Specify the formatting style (-1 for no formatting and 0 for formatting).

Return Values

Success:	Returns the size of the returned string not including the null terminator.

SetXMLElement()

This API sets a new value to the specified element in the XML.

Prototype

```
int SetXMLElement(XMLElement* rootElement, char * keyPath,
char* newValue, int bLength);
```

Parameters

rootElement	Root node structure returned by ParseXML() API
keyPath	Fully qualified path of the element whose value has to be changed.
`newValue`	New value of the element which has to be set.
`bLength`	Length of the `newValue` string

Return Values

Success:	0
Failure:	-1

WriteXMLFile()

This API writes the XML string into a file. The XML string is got from the CreateXMLStringR()API

Prototype

```
int WriteXMLFile(char *pFileName, char *pNewXMLdata, int
iDataSize);
```

Parameters

pFileName	Fully qualified path of the file to which the XML data has to be written.
pNewXMLdata	Buffer containing the XML data.
`iDataSize`	Size of the XML data.

Return Values

Success:	Returns the total number of bytes written.
Failure:	-1

Automatic Application Selection

VMAC allows multiple applications to download and execute on the terminal. FrontEnd lists all applications running on the terminal and a user can select any of the applications by pressing the function keys. This is one way to invoke an application.

It is necessary to select a particular application based on an OS event (for example, a card swipe or ICC event) when VMAC displays the idle menu. This allows for the selection of a specific application without user intervention. For example, there might be smart card application running on the terminal. When VMAC displays the idle menu of FrontEnd and a smart card is inserted for the transaction, the smart card application must be selected automatically and its idle menu displayed for any further action. This is referred to as *automatic application* selection. This is achieved by owning the smart card device and handling the `EVT_ICC1_INS` OS event to activate the application.

The smart card insertion mentioned above is just an example. An application might handle `EVT_MAG` for a card swipe, `EVT_COM2` for data on COM2, or `EVT_COM3` for data coming over a modem connected to COM3, and so on. Whatever the event might be, the steps involved to activate the application are the same. Hence an attempt is made here to define the steps in a generic way that can be used by applications for automatic selection.

The applications can be written using SDK. The applications must be VMAC enabled. There are two different ways that an application can be automatically selected in VMAC environment:

- An application can invoke itself based on an OS event. Refer to the Self Invocation section for more details.
- A separate task can invoke an application based on an OS event. Refer to the Invocation through another Task section for more details.

Self Invocation

The application must request the device on the available event with medium priority. This opens the device and waits for the OS event. On the OS event, the application requests the console device, controls the console, and displays the idle menu.

NOTE

Medium priority is recommended so that the application can release the device if another application with high priority requires the device.

This section explains the above steps using the magnetic card reader device as an example. In this example, the application name is CREDITAPP. Following is a procedure for CREDITAPP to be selected on a mag card swipe.

1. Make an entry in the resource device mapping table for a mag available event and a user-defined event to request console ownership:

```
(LOW, MAG_AVAILABLE_EVENT, MAG_GOT_EVENT, {(MAG_READER)})
(MEDIUM, REQUEST_FOR_CONSOLE, CONSOLE_GOT_EVENT, {(CONSOLE)})
```

2. Open the magnetic card reader and return to event handling mechanism in the callback for MAG_GOT_EVENT:

```
short DoMagGot()
{
   magHandle = open(DEV_CARD, 0);
   // return to event handling
}
```

3. Wait for the EVT_MAG OS event in idle state.

4. Read the mag device in this call back function and, if successful, send a REQUEST_FOR_CONSOLE event to the Device Manager.

```
short OnMagEvent()
{
   // Read the card data into a buffer
   //Send request to DEVMAN for CONSOLE. //REQUEST_FOR_CONSOLE
   EESL_send_event("DEVMAN", REQUEST_FOR_CONSOLE, NULL,0);
   //return to event handling loop.
}
```

When the console is released, CONSOLE_GOT_EVENT is received.

5. Open the console in the callback function for CONSOLE_GOT_EVENT.

After receiving the CONSOLE_GOT_EVENT, CREDITAPP continues as though the card was swiped in its idle state.

```
short Doconsole()
{
   conHandle = open(DEV_CONSOLE, 0);
   //Continue with the remaining part of transaction,
   // For ex: Display the Amount Entry dialog etc
}
```

6. Continue the transaction.

7. Release the console when the transaction is complete.

Keep the Mag device open and go back to step 3. Steps 1 to 6 must be followed to select an application based on any other OS event.

8. Replace the MAG_AVAILABLE_EVENT in the resource with the available event for the required device.

9 Change the device name `MAG_READER` to the required device.

10 Handle the corresponding OS event in the application.

Advantages

- No registration mechanism is required.
- Applications can use this approach and be invoked by more than one OS event.

Disadvantages

The situation where more than one application tries to get the same device with same priority on `AVAILABLE_EVENT`, cannot be handled. Refer to the Invocation through another Task section for more details on how this can be handled.

Invocation through another Task

This section provides details on how to achieve automatic application selection through a separate task. This separate task is referred to as `SELECT`.

NOTE

To select the application, it must be VMAC-enabled.

There are three steps involved to carry out the invocation task. It is assumed that `SELECT` can detect the application logical name of the applications to select the event to use to invoke the application and the OS event.

1 Write the resource for the device mapping table of `SELECT`.

The resource must have one record for getting the available event for the required device. There are as many records as number of devices that `SELECT` must handle.

```
(MEDIUM, //Priority
<DEVICE>_AVAILABLE_EVENT, //Input Event
<DEVICE>_GOT_EVENT, //output event
{(<DEVICE>)}) //device
```

where

- `<DEVICE>` is any device supported by Device Manager.
- `<DEVICE>_AVAILABLE_EVENT` is the available event for the device defined by the Device Manager.
- `<DEVICE>_GOT_EVENT` is the user-defined event sent by the Device Manager on getting the device.

For example, to get activated by a mag card swipe:

```
(MEDIUM, MAG_READER_AVAILABLE_EVENT,
MAG_READER_GOT_EVENT,{(MAG_READER)})
```

2 Define the logic for selecting one among several applications.

This is specific to the design of SELECT. For example, if more than one mag card-based application is available, SELECT must be able to decide which application to invoke by the track data.

3 Transfer device ownership from SELECT to the application on an OS event.

When the OS event occurs, the device must be transferred to the application. The recommended practice is to use the <DEVICE>_TRANSFER_ACTION event for the device, where device ownership can be transferred from SELECT to the application without closing the device.

Use the following procedure to transfer a device from one task to another without closing the handle. This is the most appropriate way to transfer communication devices (such as, COM1, COM2, COM3, and COM4), where the device is already open for communication by SELECT, and ownership needs to be transferred to another application without changing the settings.

a Set the owner of the device to the Device Manager:

```
short retVal;
EESL_IMM_DATA appData;
retVal = EESL_check_app_present("DEVMAN", &appData
set_owner(handle, appData.programId);
// Send the event to application through Device Manager
EESL_send_event("DEVMAN", APP_ACTIVATE_EVENT, evtData,
evtDataSize);
//The application must have the following record in its
//resource for device mapping table.
(HIGH, //priority
APP_ACTIVATE_EVENT, // Input event for application activation
APP_ACTIVATE_EVENT, // output event for application activation
{(<DEVICE>_TRANSFER_ACTION)})
```

where, APP_ACTIVATE_EVENT is the user-defined event on which the application is activated. On getting the APP_ACTIVATE_EVENT event, the application must get the handle of the device by using:

```
handle = get_owner("/dev/<DEVICE>", &taskId);
```

NOTE

It is not possible to transfer the ICC1 and ICC2 device ownership from one application to another.

In cases where SELECT involves ICC1 and ICC2 events, an alternate approach with Piggyback Events can be followed. The EESL_send_res_release_event() API is called by SELECT, which closes all devices in the list, sends RES_RELEASED_EVENT to the Device Manager, and sends the EESL event to activate the appropriate application with the specified device. EESL_send_res_release_event() ensures that the target application gets ownership of the device and no other application owns the device in between.

Use the following example to select an application with the required device.

```
short devReleased = 0;
EESL_EVENT eeslEvent;
devReleased = < DEVICE >_MASK
// if the device being released is ICC1 then
// devReleased = ICC_1_MASK
eeslEvent.cusomEESLEvent = APP_ACTIVATE_EVENT;
strcpy(eeslEvent.destinationTask, APP_NAME);
eeslEvent.evtDataLength = 0;
EESL_send_res_release_event(devReleased, &eeslEvent, NULL);
```

The application must have the following record in its resource for device mapping table.

```
(HIGH, //priority
APP_ACTIVATE_EVENT, // Input event for application activation
APP_ACTIVATE_EVENT, //output event for application activation
{(DEVICE})) // device
```

where `APP_ACTIVATE_EVENT` is the user-defined event on which the application is activated. On receiving `APP_ACTIVATE_EVENT`, the application must get the handle of the device by calling `open()` for that device. For example:

```
handle = open("/dev/device", 0);
```

Designing SELECT

`SELECT` can be designed as a background application not listed in the VMAC FrontEnd if no user interface is required. Each application that must be selected by `SELECT` must register its logical name, the OS event with which it gets selected, and the application activate event with `SELECT`. Following are a few examples to achieve this:

- `SELECT` can define an event. The applications send `SELECT` with the logical name, OS event, and activate event.
- `SELECT` can define a resource template for the logical name, OS event, and activate event. The resource file is created and downloaded along with the application. `SELECT` running in GID1 can read this information.
- `SELECT` can provide a UI where a user can enter the logical name, OS event, and activate event, or `SELECT` could request the console in case of a conflict and if it wants to display a list of applications a user can choose from.

The way `SELECT` is written depends on the choice of methods of registration listed above.

- For the first method of registration, on receiving the event from the application, enable the corresponding available event by sending `ENABLE_EVENT_ACTION` to the Device Manager.

- For the second method of registration, as SELECT reads the resource files it keeps enabling the corresponding available events by sending ENABLE_EVENT_ACTION to the Device Manager.
- For the third method of registration, when the UI is complete and correct, enable the corresponding available events by sending ENABLE_EVENT_ACTION to the Device Manager.

The following example illustrates how SELECT selects the example file CREDITAPP with a TRANSFER_ACTION event of the device on a mag card swipe.

1 Make an entry for a mag available event in the resource for the device mapping table of SELECT:

```
(MEDIUM, //Priority
MAG_AVAILABLE_EVENT, //Input Event
MAG_GOT_EVENT, //output event
{(MAG_READER)}) //Device(Mag reader) required
```

2 Open the mag card reader in the callback for MAG_GOT_EVENT and return the event handling mechanism as follows:

```
short DoMagGot()
{
   magHandle = open("/dev/mag", 0);
   // return to event handling
}
```

3 Wait for the EVT_MAG OS event in idle state.

4 Read the mag card device in this callback function and, if successful, send the activate event to CREDITAPP.

```
short OnMagEvent()
{
   unsigned char cardData[CARD_DATA_SIZE];
   short devReleased = 0;
   EESL_EVENT eeslEvent;
```

5 Read the card data into the cardData buffer.

6 Get the logical name of the application to activate on a mag card swipe.

7 Add the logic to select the application.

There might be more than one credit/debit applications, one of which must be selected based on the card data. Add all the logic to decide the name of application based on OS events, card data, and so on.

8 Get the activate event for that application.

9 Transfer device ownership to the Device Manager.

10 Send the activate event to the application along with the card data.

11 Return ownership of the device to the Device Manager:

```
{
short retVal;
EESL_IMM_DATA appData;
retVal = EESL_check_app_present("DEVMAN", &appData)
set_owner(handle, appData.programId);
// device handle - return value of device open
EESL_send_event("DEVMAN", CREDIT_ACTIVATE_EVENT, evtData,
                evtDataSize);
//return to wait_event loop
}
```

The resource for the device mapping table in `CREDITAPP` has the following records to activate event and console request events.

```
(HIGH, CREDIT_ACTIVATE_EVENT, CREDIT_ACTIVATE_EVENT,
 {(MAG_READER_TRANSFER_ACTION})),
(MEDIUM, REQUEST_FOR_CONSOLE, GOT_CONSOLE_EVENT, {(CONSOLE)})
```

`CREDIT_ACTIVATE_EVENT` is sent from the Device Manager to transfer the MAG_READER device.

12 Get the device by calling `get_owner()` in the callback function of `CREDIT_ACTIVATE_EVENT`, and send a request for console ownership:

```
short DoActivate()
{
   magHandle = get_owner("/dev/mag", &taskId);
   //Send request for CONSOLE
   EESL_send_event("DEVMAN", REQUEST_FOR_CONSOLE, NULL,0);
   //return to event handling loop.
}
```

13 Open the console in the callback function of `GOT_CONSOLE_EVENT` and display the idle menu of `CREDITAPP`.

From here, the `CREDITAPP` continues as though a card was swiped in its idle state.

```
short DoConsole
{
   conHandle = open(DEV_CONSOLE, 0);
   //Continue with the remaining steps of transaction,
   //For ex: Displaying Amount Entry dialog etc.
}
```

14 Close the mag card device after `CREDITAPP` completes the transaction and release the device:

```
short DoReleaseMag()
{
   short event = CREDIT_ACTIVATE_EVENT;
   //The event by which "CREDITAPP" got the device
   //Close the device
```

```
    close(magHandle);
    //Send RES_COMPLETE_EVENT to device manager.
    EESL_send_event("DEVMAN", RES_COMPLETE_EVENT, &event,
                    sizeof(event));
    //Display the idle menu return to event handling loop.
}
```

The Device Manager releases the mag card device to `SELECT` on a `MAG_AVAILABLE_EVENT`, as illustrated in step 2, and the remaining steps are repeated.

Advantages

- A separate task handles the OS events.

Disadvantages

- A registration method must be defined by `SELECT`.
- It is not possible to select more than one application based on the same OS event.
- More system memory is used by this new task.

Using a Contactless Device

Unlike the magnetic card reader which has an OS event `EVT_MAG`, a contactless device has no specific event to generate when data is available for the application to read. For the contactless device to generate an event, the application must use the OS function `set_event_bit()` and specify which of the unused OS events should be assigned to report the contactless event.

The example below illustrates the use of a contactless device. The application LOYALTYAPP is activated upon tapping a card on the device.

1. Make an entry in the resource device mapping table for a CTLS available event and a user-defined event to request console ownership:

```
(LOW, CTLS_1_AVAILABLE_EVENT, CTLS_GOT_EVENT, {(CTLS1)})
(MEDIUM, REQUEST_FOR_CONSOLE, CONSOLE_GOT_EVENT,
{(CONSOLE)})
```

2. Open the contactless device and return to the event handling mechanism in the callback for `CTLS_GOT_EVENT`.

```
short DoCTLSGot()
{
ctlsHandle = open(DEV_CTLS, 0);
// return to event handling
}
```

3 Determine which predefined OS event is unused. In this example, the `LOYALTYAPP` will be installed in terminals that have no barcode readers so the `EVT_BAR` event is assigned to the contactless event.

```
#define MY_EVT_CTLS EVT_BAR
short DoCTLSGot()
{
ctlsHandle = open(DEV_CTLS, 0);
set_event_bit(ctlsHandle, MY_EVT_CTLS);
// return to event handling
}
```

4 Wait for `MY_EVT_CTLS` OS event in idle state.

5 Read the contactless device in this callback function and, if successful send a `REQUEST_FOR_CONSOLE` event to the Device Manager:

```
short OnCTLSEvent()
{
// Read the card data into a buffer
//Send request to DEVMAN for CONSOLE. //
REQUEST_FOR_CONSOLE
EESL_send_event("DEVMAN", REQUEST_FOR_CONSOLE, NULL,0);
//return to event handling loop.
}
```

6 Follow steps 5-10 in the `CREDITAPP` (Self Invocation) example.

VMAC Interface (VMACIF)

The VMAC Interface (VMACIF) is a bridge application working between VMAC and Verix EOS components. VMAC's device management function is the focus of VMACIF.

When an application (the requesting application) needs a device, e.g., the printer, it requests VMAC. In turn, VMAC requests the printer from the application that owns it (the owning application) and hands it over to the requesting application. The transfer of the device from the owning application to the requesting application is managed by VMAC.

NOTE Both the owning and requesting applications are VMAC-compliant, i.e., both applications honor requests from VMAC and respond to it appropriately. Both applications function in a co-operating environment and are seamlessly able to share devices. Should any one application fail to co-operate, the sharing of devices fails and consequently all applications start to malfunction.

VMAC is designed to "grab" all known devices and hold on to them at start up, building up its inventory of devices. When a device is given to an application, it is "checked out," allowing VMAC to know which application has control of which device(s). When another application requests for the device, it follows the steps described earlier. Applications may also release a device to VMAC on their own volition.

NOTE If VMAC is unable to "grab" or obtain a device at start up, that application does not get listed in the device inventory. Since all applications depend on VMAC to obtain devices, a device missing from the inventory is by definition unavailable to all other applications.

The VMACIF application works with the Verix EOS Communication Engine (CommEngine) program (VXCE.OUT) and the Network Control Panel (NCP) (VXNCP.OUT) program which are resident in Verix EOS. Verix EOS starts up before any application in GID 1-15, as do CommEngine and NCP. If CommEngine were to start up the communication infrastructure, the first task it would perform is to grab or open the communication devices (or modems). Once a device is open, VMAC cannot obtain it and it cannot be inventoried. Conversely, neither CommEngine nor NCP are VMAC-compliant applications; they do not register with VMAC and cannot handle device requests.

These are conflicting requirements that must be met by VMAC, CommEngine and NCP. Application VMACIF works with VMAC, CommEngine and NCP; and, by doing so, it satisfies the requirements of both VMAC and Verix EOS components.

Startup Operation

CommEngine Startup Operation

CommEngine is designed to bring up the network on startup. One of the first tasks of CommEngine is to grab and open the communication hardware. This act of opening the device causes CommEngine to own the device.

CommEngine needs to know if it is running in a VMAC environment or more generally, a device-managed environment. At startup, CommEngine looks for parameter *CEDM (CommEngine Device Managed) in GID1.

- If *CEDM is present and set to 1 (*CEDM=1), CommEngine will work with VMACIF to obtain the necessary communication device.
- If *CEDM is not present or set to zero (*CEDM=0), CommEngine will work without VMACIF and will proceed with opening and owning the communication devices.

If configuration parameter *CEDM is set (i.e. device management is enabled) CommEngine will wait for a certain duration for an application to register. At the end of the duration, CommEngine will proceed to open the device and proceed as though *CEDM is not present.

Network Control Panel Startup Operation

VeriFone designed NCP as the single user interface (UI) for all Verix EOS components. By default, NCP starts up and waits until it gets activated by any application via the ceAPI call ceActivateNCP(). Once NCP has initialized and completed registration/connection with other components, it will remain “dormant,” waiting to be activated via ceAPI.

In a multi-application environment, switching UI between applications is possible via HotKey and VMAC menus. VeriFone recommends not modifying applications to use ceAPI, and instead invoking NCP via the VMAC menu. Since NCP runs before VMAC, NCP cannot register with VMAC/IMM, so NCP is not aware of

VMAC, due to the sequence of execution, and VMACIF interfaces with other applications on NCP's behalf. Whenever a user selects VMACIF from the VMAC menu, it will activate NCP immediately. Since VMACIF performs no screen operations, it appears seamless.

VMACIF Startup Operation

Application VMACIF.OUT is started by VMAC and on start up performs these three tasks in the following sequence:

1. Registers with VMAC using VMAC's EESL API EESL_Initialise(). The VMAC logical name of the application is VMACIF.
2. Registers with CommEngine using ceAPI.
3. Registers with CommEngine as its device management application.

NOTE

The VMACIF application may be running in any GID (1 to 15). However, if located in GID1, it can review parameter *CEDM and exit if it is not present or equal to zero (0).

VMACIF Interaction with Verix EOS

CommEngine-VMACIF Interaction

VMACIF registers with CommEngine as CommEngine's device management application. At this point, CommEngine operates on a device push and pull model. When the device is available, VMACIF pushes the device to CommEngine. Similarly when VMACIF needs the device, it pulls the device from CommEngine. Since VMACIF is registered as the CommEngine device management application, CommEngine is primed for such action by VMACIF.

Device Push (VMACIF → CommEngine)

When VMACIF obtains/assigns the device from VMAC, it pushes the device to CommEngine using ceAPI ceReleaseDevice(), releasing the device to CommEngine. Once CommEngine has the device, it opens and owns the device, starts the network connection, and continues until VMACIF makes a request for the device.

Device Pull (CommEngine → VMACIF)

When another application needs the communication device, it makes this request via VMAC, and VMAC, in turn, requests VMACIF for the device. In VMAC's device allocation table, the communication device is checked out to VMACIF for use by CommEngine. VMACIF pulls the device using ceAPI ceRequestDevice().

Network Control Panel-VMACIF Interaction

VMACIF masquerades as NCP and registers with VMAC, creating a menu option titled NCP and making itself selectable from VMAC.

NOTE

It is important for NCP to return the CONSOLE to the same application that triggered its activation. For this purpose, NCP will keep ownership of the CONSOLE device by disabling hot key until the user explicitly selects “Exit” from the UI.

VMACIF Invoking NCP

When the user selects NCP from VMAC menu, VMAC transfers control of console and printer devices to VMACIF. VMACIF immediately transfers control of these devices to NCP using ceAPI – ceActivateNCP().

NCP returning to VMACIF

When the user exits NCP, control of console and printer devices are transferred to VMACIF, immediately relinquishing control of these devices and transferring the control back to VMAC.

Application Packaging

VMACIF binaries can be found under the “VMAC” directory.

Directory Structure

Figure 9 shows the VMACIF directory structure.

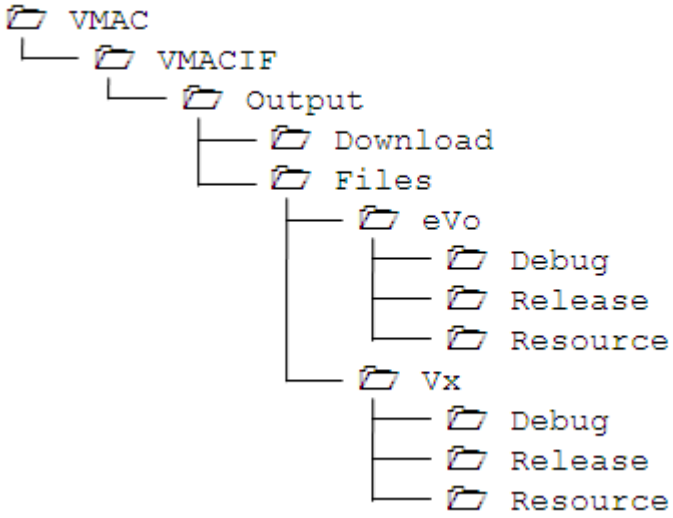


Figure 9 VMACIF Directory Structure

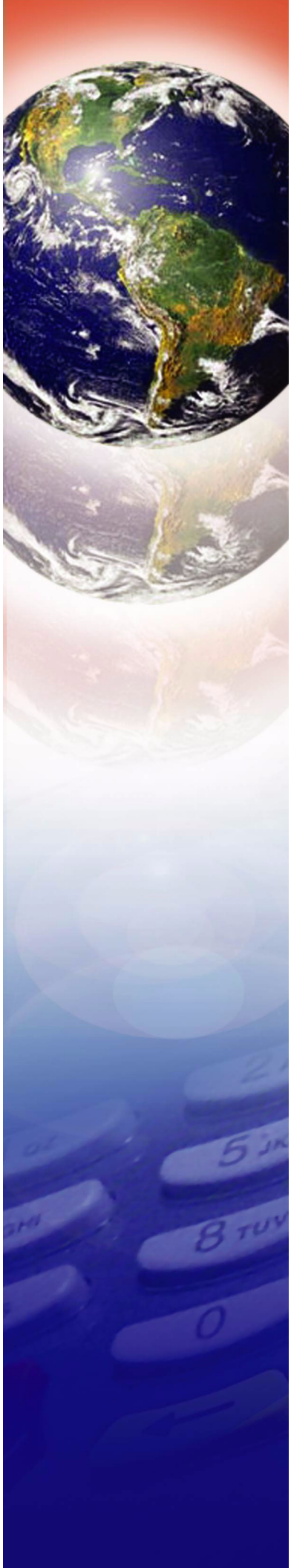
Directory Contents

Table 13 describes the contents of VMACIF directories.

Table 13 VMACIF Directory Contents

Directory	Description
Download	Contains dlVMACIF.bat and SignVMACIF.bat for downloading and signing VMACIF binaries.
eVo	VMACIF binaries compiled using Verix eVo VMAC.
Vx	VMACIF binaries compiled using Verix V VMAC.
Debug	Contains the debug version of VMACIF.
Release	Contains the release version of VMACIF.
Resource	Contains the resource files (.res) needed by VMACIF

VeriFone, Inc.
2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA
(800) VeriFone (837-4366)
www.verifone.com



Verix eVo Multi-App Conductor

Programmers Guide

VeriFone Part Number DOC00306, Revision A