VINETIC® Driver

Voice and Internet Enhanced Telephony Interface Circuit

VINETIC®-4M (PEF 3314) Version 2.1/2.2
VINETIC®-4C (PEF 3394) Version 2.1/2.2
VINETIC®-4S (PEF 3304) Version 2.1
VINETIC®-4VIP (PEB 3324) Version 1.4
VINETIC®-2VIP (PEB 3322) Version 1.4
VINETIC®-0 (PEB 3320) Version 1.4
VINETIC®-2CPE (PEB 3332) Version 1.4

Wireline Communications

Never stop thinking.

Edition 2005-07-27

Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
81669 München, Germany

Attention please!

The information herein is given to describe certain components and shall not be considered as a guarantee of characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Previous Version:

Page	Subjects (major changes since last revision)

Trademarks

Table of Contents

Preface

This document, the VINETIC® Driver Porting and Integration Guide Rev. 1.0, is valid for the following VINETIC® codecs:

VINETIC®-4M (PEF 3314) Version 2.1/2.2
VINETIC®-4C (PEF 3394) Version 2.1/2.2
VINETIC®-4S (PEF 3304) Version 2.1
VINETIC®-4VIP (PEB 3324) Version 1.4
VINETIC®-2VIP (PEB 3322) Version 1.4
VINETIC®-0 (PEB 3320) Version 1.4
VINETIC®-2CPE (PEB 3332) Version 1.4

The VINETIC® driver is a platform independent driver software consisting of an operating system layer and an operating system independent generic part. This driver is intended to run on each system integrating the VINETIC® chip. Therefore, this document is (only) intended to guide the porting as well as the integration of the VINETIC® driver on a new system under consideration of its operating system and its hardware platform.

1 Porting to Operating System

This chapter addresses porting issues related to the used operating system.

1.1 Operating System Macros

A super set of operating system macros is defined and used throughout the VINETIC® driver, making the driver operating system independent. **Table 1** lists these operating system macros abstractions and gives a rough overview. For more details, refer to appropriate header file which is part of the source code (see **Chapter 1.2**).

Table 1 Operating Systems Macros

Name	Description
Hardware Macros, defined in Board Support Package (BSP).	
IFXOS_UC_BASE	Returns the microprocessor base address.
__BYTE_ORDER	Defines the microprocessor endianess which will be considered in the driver: __LITTLE_ENDIAN for a little endian and __BIG_ENDIAN for a big endian system.
Memory Management Macros	
IFXOS_MALLOC(...)	Allocates memory.
IFXOS_FREE(...)	Frees memory.
IFXOS_CPY_USR2KERN(...)	Copies data from user to kernel space.
IFXOS_CPY_KERN2USR(...)	Copies data from kernel to user space.
Interrupt Management Macros	
IFXOS_INTSTAT	Interrupt status data type.
IFXOS_LOCKINT(...)	Locks interrupt handling.
IFXOS_UNLOCKINT(...)	Unlocks interrupt handling.
IFXOS_IRQ_DISABLE(...)	Disables interrupt.
IFXOS_IRQ_ENABLE(...)	Enables interrupt.
Time Management Macros	
IFXOS_Wait(...)	Delays execution with task schedule.
IFXOS_DELAYMS(...)	Short active delay in milliseconds without schedule.
IFXOS_DELAYUS(...)	Short active delay in microseconds without schedule.
Event Type and Event Handling Macros Events are used for the communication between high priority tasks or interrupt and other tasks (e.g: to signalize a task sleeping on an event that the event has occurred).	
IFXOS_event_t	Event data type.
IFXOS_WAIT_FOREVER	Waits forever.
IFXOS_NOWAIT	Never waits.
IFXOS_InitEvent(...)	Initializes an event.
IFXOS_WakeUpEvent(...)	Signals an event.
IFXOS_ClearEvent(...)	Resets an event to the initial state.
IFXOS_WaitEvent_timeout(...)	Waits for a specified event with a specified timeout to occur or timed out.
IFXOS_WaitEvent(...)	Waits for a specified event with a specified condition to occur or timed out.
Mutex Type and Mutex Macros Mutexes are used to protect critical sections against race conditions. They have several names across operating systems, but are all considered as mutexes by the VINETIC® driver.	

Table 1 Operating Systems Macros (cont'd)

Name	Description
IFXOS_Mutex_t	Mutex data type.
IFXOS_MutexInit(...)	Initializes mutex.
IFXOS_MutexLock(...)	Locks/takes the mutex.
IFXOS_MutexUnlock(...)	Unlocks/Gives the mutex.
IFXOS_MutexDelete (...)	Deletes a mutex element.
Selecting and Polling The poll/select mechanism is used for user application synchronization after occurrence of particular events (e.g: signalization to application, that data is ready for reading)	
IFXOS_wakelist_t	Wakeup data type for select wait queues.
IFXOS_Init_WakeList(...)	Initializes a queue.
IFXOS_SleepQueue(...)	Initializes the sleep on a given queue.
IFXOS_WakeUp(...)	Wakes up a waiting queue in poll/select.
IFXOS_WRITEQ	Defines a write queue.
IFXOS_READQ	Defines the read queue.
IFXOS_SYSWRITE	Flag which signalizes that the system event for write is ready.
IFXOS_SYSREAD	Flag which signalizes that the system event for read is ready.

Nevertheless, some operating system files must be ported or implemented for the VINETIC® driver to be fully operational with the used operating system. This is the topic of **Chapter 1.2**.

1.2 Operating System Files

This chapter addresses the operating systems files which must be adapted or implemented for a full operating system compatibility.

1.2.1 Macros Adaptation File

The operating system macros listed in **Chapter 1.1** are all implemented in a central header file called **<sys_drv_ifxos.h>**, where they are mapped to the appropriate operating system calls to insure the functionality behind them. This header file is part of the released source code and must be adapted for any new, yet unsupported operating system.

Under **Linux®** and **VxWorks®**, theses macros are fully integrated and supported. They essentially map operating system specific types or functions.

Therefore, wrappers must be implemented in this file in case the used operating system doesn't support any of the functionalities behind the macros and types defined in **Chapter 1.1** (e.g: wrapper for events, mutexes, poll/select).

1.2.2 VINETIC® Driver Operating System File

The operating system interface must be implemented in a file called **drv_vinetic_<os>.c**[1]. This file represents the operating system abstraction layer of the VINETIC® driver. Under **Linux®** and **VxWorks®** operating systems (which are currently supported), this file implements the common known UNIX-like interface open/close/ioctl/read/write and a select mechanism, as well as an entry and an exit function for the registration and unregistration of the driver.

1) <os> is the placeholder for the name of the operating system (e.g. drv_vinetic_linux.c, drv_vinetic_vxworks.c)

Attention: The select mechanism is used to support non blocking I/O operations (e.g.: read/write) between an application and the underlying driver. This mechanism is fully supported under Linux® (poll method) and VxWorks® (select method) operating systems. In case the used operating system doesn't support this mechanism, an emulation must be implemented.

2 Porting to Hardware System

This chapter addresses hardware related issues to ensure that the VINETIC® driver runs on it without problems.

Attention: This is not a hardware integration guide. For VINETIC® chip related hardware integration, refer to the VINETIC® Data Sheets and the Hardware Design Guide [20] as listed on the Page 27.

2.1 Clocking Considerations

There are specific clocking requirements according to the VINETIC® chip version actually used. These requirements are fully specified in the appropriate VINETIC® Data Sheet where following main requirements are stated:

- VINETIC®-4VIP/-2VIP/-0/-2CPE Version 1.4 needs at least three clocks to work properly: the master clock (MCLK), the frame synchronization clock (FSC) and the PCM Interface clock (PCL). These three clocks are mandatory and must be provided at all times regardless of the application to ensure the operation of the VINETIC® device. Also, the hardware designer must make sure that the master clock (MCLK) is phase-locked to PCM Interface clock (PCL) and the frame synchronization FSC, whereby it is recommended to directly connect MCLK and PCL.
- VINETIC®-4M/-4C Version 2.1/2.2 and VINETIC®-4S Version 2.1 require the continuous presence of the frame synchronization clock (FSC) and the PCM Interface clock (PCL) to have the VINETIC® device operation ensured. Also, the hardware designer must make sure that the PCM interface clock (PCL) is phase-locked to the frame synchronization (FSC)

Clock synchronization problems are reported by the VINETIC® chip hardware status register (HWSR1) and are of two categories:

- Clock divider synchronization failure: SYNC-FAIL
- PLL synchronization failure: MCLK-FAIL

Note: The clock settings must be done properly before using the VINETIC® driver. The VINETIC® chip will not work properly if there are clock problems and therefore also not the VINETIC®driver which interacts with the VINETIC®.

2.2 Reset Considerations

A hardware reset of the VINETIC® chip is initiated by a power-on reset or by a hardware reset. A hardware reset requires setting the signal at the RESETQ input pin to low-level for at least 4 µs. The reset input pin has a spike rejection that will safely suppress spikes with a duration of less than 1 µs. By pulling the RESETQ input pin to low, the chip will be reset (see **Figure 1**) and the following actions will take place:

- All I/O pins are deactivated
- All outputs are inactive (e.g. DX1/DX2)
- The internal PLL is stopped
- Internal clocks are deactivated
- The chip is in a Reset State

With the rising edge of the reset signal all external clocks need to be already stable and then the following actions will take place:

- Clock detection
- PLL synchronization
- Execution of the reset routine: after the reset routine has finished, an interrupt is generated and the RESET bit in the ISR register is set
- The EDSP stays in boot state
- The ALM Modules (Analog Channels) are in PDH (Power Down High Impedance) mode.

The internal reset routine requires approximately 12 frames (12 × 125 µs = 1.5 ms) to be finished (including PLL start-up and clock synchronization). First access to the VINETIC® is possible after the INTQ signal = 0.

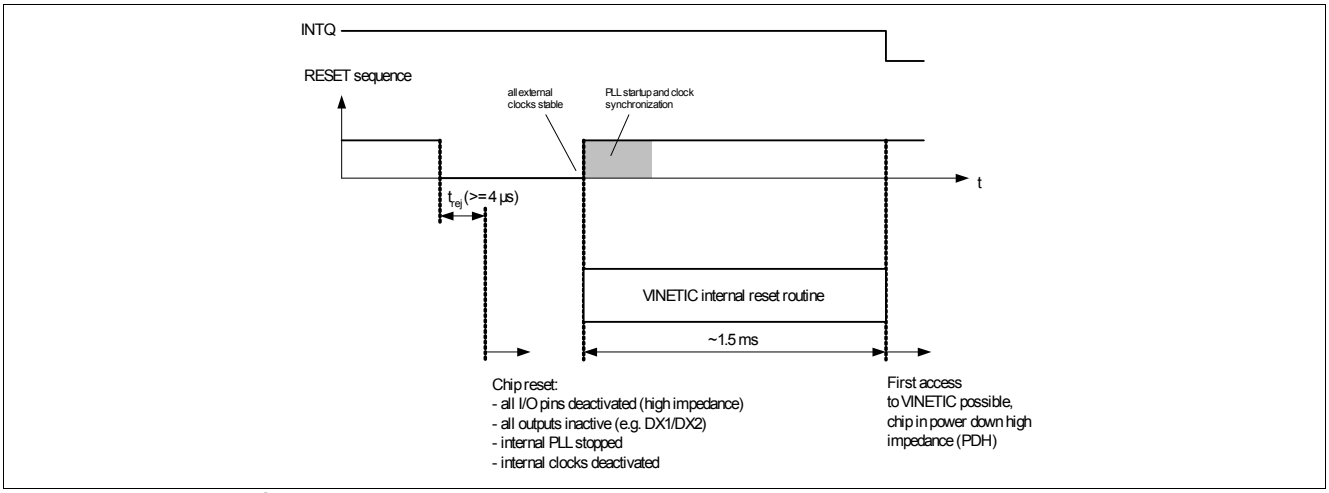

Figure 1 VINETIC® Reset Sequence

Therefore, it is mandatory to respect the reset active time (at least 4 µs) and the reset inactive time (at least 1.5 ms), otherwise the correct operation of the VINETIC® chip can not be guaranteed. The recommended software flow is depicted in **Figure 2**.

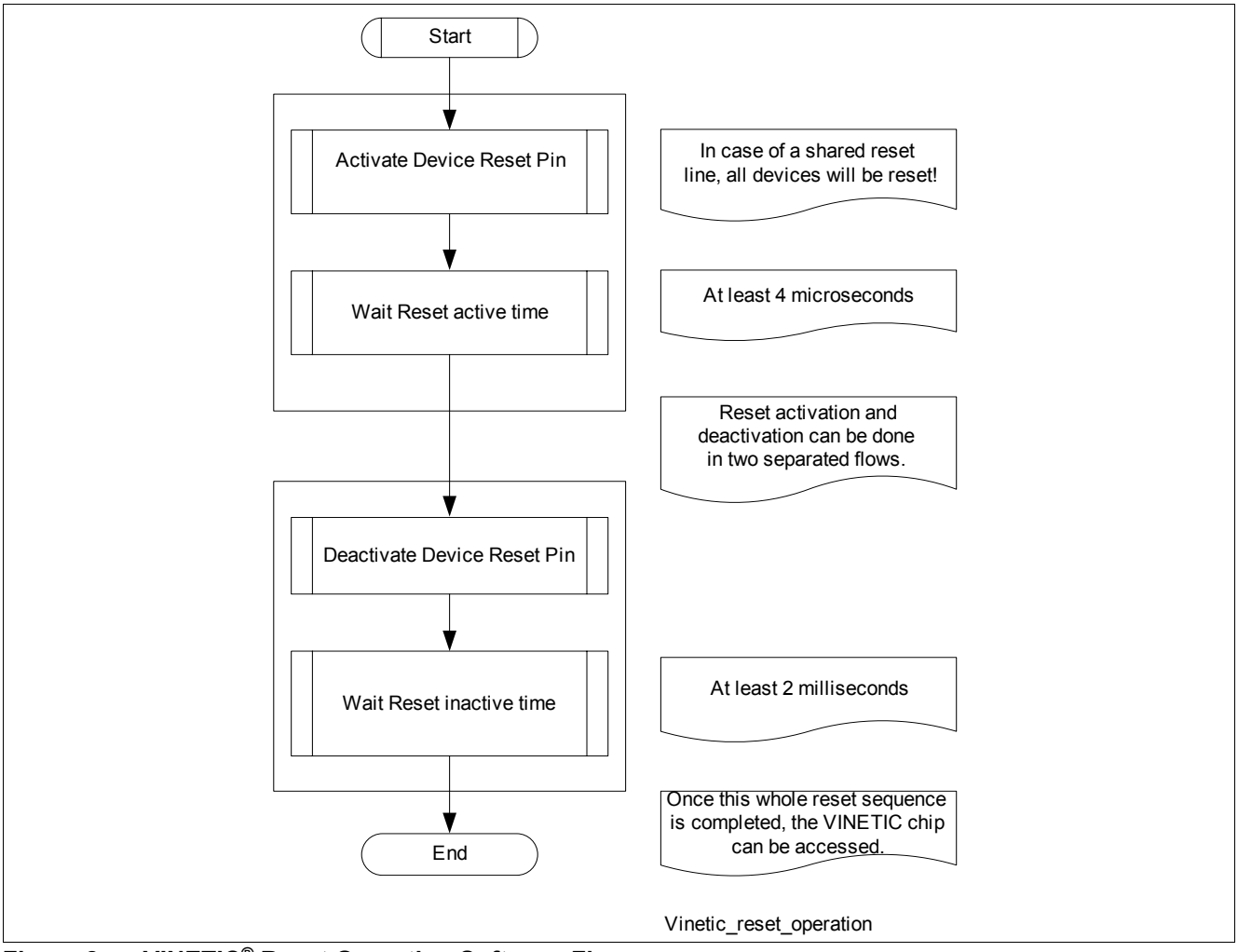

Figure 2 VINETIC® Reset Operation Software Flow

Note: The VINETIC® driver does not provide a VINETIC® hardware reset functionality. The implementation of this operation is left to the system integrator.

2.3 Endianess Considerations

The operating system header file which contains the endianess information (little/big endian) must be included in the file <***sys_drv_ifxos.h***> and the hardware generic macro __BYTE_ORDER must be set either to __LITTLE_ENDIAN or to __BIG_ENDIAN according to the used endianess (see **Chapter 1.1**). This setting is important and required for the handling of 8-bit data which should be converted in other data types (e.g: 8-bit to 16-bit, 8-bit to 32-bit).

Attention: It is mandatory to update the file <sys_drv_ifxos.h> in case the used operating system is not yet supported.

2.4 Access Mode Considerations

The digital interfaces of the VINETIC® are operated by a programmable host interface controller and allow flexible and easy adaptation to various interfaces. For programming the VINETIC® and performing data/packet transfer to/from the VINETIC®, a parallel interface or a serial micro controller interface ca be used. Additionally, the VINETIC® is equipped with a PCM data interface.

The parallel interface can be operated in 8/16-bit Intel mode (multiplexed/de multiplexed) or in 8/16-bit Motorola mode.

The VINETIC® serial micro controller interface (µC interface = SCI) is compatible with the Motorola SPI.

The VINETIC® PCM interface has two PCM highways and can be operated together with the serial µC interface or the parallel interface.

With these access interfaces, the VINETIC® device supports the widely used micro controllers: e.g. MPC850, MPC860, MPC8260, C165UTAH, MIPS and ARM derivatives, etc. All parallel and serial interfaces (host interfaces) use the same (multiplexed) pins. The desired interface type is selected by means of pin strapping with the pins IFSEL0, IFSEL1, IFSEL2 and IFSEL3 (IFSEL3 only with some specific VINETIC® pin packages). Therefore, it is up to the hardware designer either to set the access mode (fixed pin settings) or by using a logic (e.g. CPLD) between the micro controller and the VINETIC®. Refer to VINETIC® Data Sheets (see **Page 28**) for more details about VINETIC® interfaces.

*Note: The VINETIC® driver expects the selected access mode via a dedicated interface for its internal mappings at initialization time (see **Chapter 3.3.1**). To support the serial micro controller interface (SPI), the VINETIC® driver requires the implementation of specific macros in its user configuration file (see **Chapter 2.9** and **Chapter 3.3.3.2**).*

Attention: Regardless whether the PCM Interface is used or not, all clock sources (as described in Chapter 2.1) must be provided all the time to ensure the correct operation of the VINETIC® device.

Table 2 shows which access modes are allowed on different VINETIC® versions and specifies the action to take to support this access mode.

Table 2 Access Mode Table

Access Mode	Version	Remark
Motorola 16-bit	VINETIC®-4M/-4C/-4S V**2.x** VINETIC®-4VIP/-2VIP/-0/-2CPE V**1.4**	–
Intel 16-bit Multiplexed	VINETIC®-4M/-4C/-4S V**2.x**	Does not work for V**1**.x (see **[22]**)
Intel 16-bit Demultiplexed	VINETIC®-4M/-4C/-4S V**2.x** VINETIC®-4VIP/-2VIP/-0/-2CPE V**1.4**	–

Table 2 Access Mode Table (cont'd)

Access Mode	Version	Remark
Motorola 8-bit	VINETIC®-4M/-4C/-4S V**2.x** VINETIC®-4VIP/-2VIP/-0/-2CPE V**1.4**	V1.4 requires PHI download (see **[18]**)
Intel 8-bit Multiplexed Big Endian	None	See **[22]**
Intel 8-bit Demultiplexed Big Endian	VINETIC®-4VIP/-2VIP/-0/-2CPE V**1.4**	V1.4 requires PHI download (see **[18]**)
Intel 8-bit Multiplexed Little Endian	VINETIC®-4M/-4C/-4S V**2.x**	–
Intel 8-bit Demultiplexed Little Endian	VINETIC®-4M/-4C/-4S V**2.x**	–
SCI (SPI Mode)	VINETIC®-4M/-4C/-4S V**2.x** VINETIC®-4VIP/-2VIP/-0/-2CPE V**1.4**	–

2.5 Interrupt Considerations

The hardware designer connects the VINETIC® interrupt line to the used micro controller. Therefore, the interrupt line number used by the micro controller must be communicated to the VINETIC® driver, so that it can register its interrupt routine. This is done during the VINETIC® driver initialization time (see **Chapter 3.3.1**).

Actually, the VINETIC® driver assumes that the interrupts are level-triggered. Therefore, it does not provide any acknowledgement as needed by edge-triggered interrupts.

Attention: It is strongly recommended to use level-triggered interrupt to avoid losing interrupts while the line is disabled, which often happens with edge-triggered interrupts.

By default, the VINETIC® driver uses the operating system calls for interrupt operations (e.g. register/enable/disable/unregister interrupts). Nevertheless, for systems which implement a logic device for the interrupt handling (e.g. FPGA controlling shared interrupt line), the VINETIC® driver provides a set of macros which must be adapted for this purpose (see **Chapter 2.8** and **Chapter 2.9**).

2.6 SLIC Considerations

As it is up to the hardware designer to choose which SLIC[1)] chips are suitable for the system being designed, this chapter addresses the SLIC usage dependent (software) considerations.

2.6.1 Line modes

The supported line modes are dependent on the particular SLIC module used. For some SLICs, not all line modes are supported as specified in **[19]**. Refer to **[9]**, **[10]** and **[12]** for more details.

2.6.2 CRAM Coefficients

CRAM coefficients (*.BYT files) are SLIC dependent. They must be calculated using the VINETICOS software (available on the VINETIC® Tool Package CD) and are downloaded channel-wise or device-wise (broadcast) to the VINETIC® with the interface FIO_VINETIC_DOWNLOAD_CRAM or during the VINETIC® TAPI initialization. CRAM coefficients download is actually a user task. Therefore, the driver doesn't download any default CRAM coefficients automatically.

Note: Please take care to select the appropriate BYT file for the used hardware.

1) SLIC = Subscriber Line Interface Circuit

Example

```
VINETIC_IO_CRAM ioCram;
IFX_int32_t ret;

memset(&ioCram,0, sizeof (VINETIC_IO_CRAM));
ioCram.bBroadCast   = 1;
ioCram.nFormat      = VINETIC_IO_CRAM_FORMAT_2_1;
ioCram.nStartAddr   = VINETIC_COP_START_ADDRESS;
ioCram.nLength      = sizeof (VINETIC_COEFF)/2;
ioCram.nCRC         = VINETIC_CRAM_CRC;
ioCram.bcr1.nData   = VINETIC_COEFF_BCR[0][0];
ioCram.bcr1.nMask   = VINETIC_COEFF_BCR[0][1];
ioCram.bcr2.nData   = VINETIC_COEFF_BCR[1][0];
ioCram.bcr2.nMask   = VINETIC_COEFF_BCR[1][1];
ioCram.tstr2.nData  = VINETIC_COEFF_TSTR2[0][0];
ioCram.tstr2.nMask  = VINETIC_COEFF_TSTR2[0][1];
memcpy(ioCram.aData, VINETIC_COEFF, sizeof (VINETIC_COEFF));

ret = ioctl(fdVinChan, FIO_VINETIC_DOWNLOAD_CRAM,(int)&ioCram);
```

2.7 Multiple VINETIC® Chips Support

The VINETIC® driver is designed to support several VINETIC® devices at once. Therefore, the VINETIC® device number must be provided using the compiler macro VINETIC_MAX_DEVICES:

`-DVINETIC_MAX_DEVICES=<system device number>.`

The value is set by default to 1.

In case that the driver must support more than one VINETIC® device, shared interrupts and reset lines come into consideration. These topics are discussed in the following chapters.

Attention: It is mandatory to specify how many VINETIC® devices are on the system being integrated when compiling the VINETIC® driver for that system. Otherwise, the VINETIC® driver assumes that the system has only one VINETIC® device and also supports only one device.

2.7.1 Shared Interrupt Concept

If several VINETIC® devices are connected to only one micro controller interrupt line, the VINETIC® driver provides shared interrupt support for **Linux®** and **VxWorks®** operating systems. If instead the shared interrupt line is controlled by a logic device (e.g. FPGA device), the VINETIC® driver provides a set of macros which must be adapted accordingly (see **Chapter 2.5** and **Chapter 2.9**).

Attention: The shared interrupt support implementation must be done for all currently unsupported operating systems (see Chapter 1.2.2 for more details).

2.7.2 Shared Reset Line

If the reset line is shared by several VINETIC® devices, all these chips will be reset when the reset pin is activated and deactivated (see **Chapter 2.2**). This must be taken into account in the system software design, which must provide a mechanism to reset each VINETIC® device separately without influence on the other running devices.

2.8 Other System Considerations

This chapter comprises some VINETIC® driver system considerations which make it necessary to implement a system abstraction layer file in addition to the already provided VINETIC® driver interfaces for the integration. The reason for this is that these system considerations could not be addressed by means of the provided interfaces.

Considering system differences, a list on several systems usable (optional) system macros was defined.

The driver uses these macros for special system parameters which can be overruled by a user configuration file (see **Chapter 2.9**).

This ensures the flexibility needed by the VINETIC® driver which has to support several system implementations without increased complexity.

List of the (optional) system macros (set at compile time):

- PARACCESS_CS_RECOVER(...)
  This macro is intended for systems which need a specific chip select recovery after each chip access (e.g: AM5120). For other systems, the macro is by default empty.
- VIN_DISABLE_IRQLINE(...)/ VIN_ENABLE_IRQLINE(...)
  These macros are intended to map the enable/disable IRQ routines for systems not using the operating system methods for this action (e.g. FPGA device controls interrupt handling). They are by default mapped to the appropriate operating system routines.
- VIN_SYS_REGISTER_INT_HANDLER(...) / VIN_SYS_UNREGISTER_INT_HANDLER(...)
  These macros are intended for the registration / deregistration of the interrupt handler in case that the operating system routines aren't suitable for this purpose (e.g. user implements his own assembler routines for theses purposes). They are by default mapped to the operating system routines (see **Chapter 1.1**).
- VIN_1X_PARACC_MUX_SHIFT / VIN_2X_PARACC_MUX_SHIFT / VIN_PARACC_SHIFT
  Theses macros define the (<<) shift factor to apply to the VINETIC® chip access addresses NWD, EOM and DIA in multiplexed and in non multiplexed mode. They are adaptation factors for the microprocessor addressing (e.g: If it is not possible to access odd addresses on a specific microprocessor, the factor will be set to 1, so that only even addresses are accessed). These macros are also part of the user configuration header file and can be adapted by each system when necessary.

2.9 VINETIC® Driver System Configuration File

The VINETIC® driver provides a system abstraction layer header file in which several (optional) system specific macros can be redefined if needed to ensure full support of the system being integrated (e.g: Macros defined in **Chapter 2.8**).

This file, called <***drv_config_user.h***>, is system specific (which means not common) and therefore must be located in the system build directory.

The VINETIC® driver considers the macros defined in this file only if compiled with the specific compiler switch '*-DENABLE_USER_CONFIG*'.

Note: Under Linux®, this macro is set when the argument --enable-user-config is passed to the configure/autogen script.

A default template of this file, called <***drv_config_user.default.h***>, is available with the released source code. This file contains system macros which can be adapted accordingly. Once adapted, the file must be placed in the target build directory and renamed to <***drv_config_user.h***>.

A direct application of this file is the system dependent SPI support (see **Chapter 2.4** and **Chapter 3.3.3.2**).

Table 3 shows a rough overview of system relevant macros defined in the file. For more details (e.g: about macros parameters) please refer to the file itself.

Table 3 System optional Macros

Name	Description
Error Setting Macro	
SET_ERROR(...)	Macro to signal and set an error. Useful to generate a trigger signal during hardware debugging.
Ready Flag Timeout (only relevant for VINETIC®-4VIP)	

Table 3 System optional Macros (cont'd)

Name	Description
WAIT_RDY	Number of accesses to ready flag (only relevant for VINETIC®-4VIP). Optimization to minimum of processor access time. Default value is 500.
SPI Access Support Macros (usage enabled with -DVIN_SPI)	
SPI_MAXBYTES_SIZE	SPI buffer size in bytes (8-bit) according to the SPI driver.
SPI_CS_SET(...)	Macro to set/unset SPI chip select.
spi_ll_read_write(...)	Macro mapping the low-level SPI read/write routine, exported for example by an SPI driver.
Parallel Access Support Macros	
PARACCESS_CS_RECOVER	Chip select recovery after a micro controller interface access, as needed by some platforms using the MIPS-4KC controller (e.g. AM5120)
VIN_1X_PARACC_MUX_ SHIFT	Address shift factor (<<) needed for a VINETIC®-4VIP/-2VIP/-0/-2CPE Version 1.4 parallel intel mux access, dependent on the data bus connection and the processor architecture. Default is 0, meaning no shift. Access done only on even addresses (NWD = 0x02, EOM = 0x04).
VIN_2X_PARACC_MUX_ SHIFT	Address shift factor (<<) needed for a parallel intel mux access on VINETIC®-4M/-4C Version 2.1/2.2 and VINETIC®-4S Version 2.1, depends on the data bus connection and the processor architecture. Default is 1, meaning access only through even addresses: NWD=0x02 and EOM=0x03 are shifted.
VIN_PARACC_SHIFT	Address shift factor (<<) needed for a VINETIC® parallel access, exclusive intel mux access, depends on the data bus connection and the processor architecture. Default is 0 (no shift). Access possible on even (NWD = 0x02) and odd (EOM = 0x03) addresses.
Interrupt Operations Support Macros (in case OS methods aren't suitable)	
VIN_DISABLE_IRQLINE(...)	Macro to disable the interrupt line specified, by default set to OS method (see **Chapter 1.1**).
VIN_ENABLE_IRQLINE(...)	Macro to enable the interrupt line specified, by default set to OS method (see **Chapter 1.1**).
VIN_SYS_REGISTER_INT_ HANDLER(...)	Macro which maps the function taking care of the interrupt handler registration, by default set to OS method implemented in OS file (see **Chapter 1.2**).
VIN_SYS_UNREGISTER_INT_ HANDLER(...)	Macro which maps the function taking care of the interrupt handler unregistration, by default set to OS method implemented in OS file (see **Chapter 1.2**).

3 Integrating the VINETIC® Driver

This chapter addresses the issues which must be considered when integrating the VINETIC® driver on a specific system.

3.1 Relevant Compiler Options

Table 4 shows the set of VINETIC® driver relevant compiler switches which can be used when compiling the VINETIC® driver. Operating system specific switches are not described here (refer to the operating system manuals).

A template configuration file (drv_config.h.in) comprising all relevant compiler switches is part of the VINETIC® driver source code distribution and can be used as example. Refer to the VINETIC® driver readme file located in the driver top directory for more details.

Note: For the Linux® operating system, the switches described here are mapped to configure / autogen.sh options. After generation of the Makefile, a driver configuration file, called drv_config.h, is generated in the build directory. It contains the set of compiler settings used for the built target.

Table 4 VINETIC® Driver Compiler Options

Name	Description
-DVIN_8BIT	Enables 8-bit bus access support in the VINETIC® driver.
-DVIN_SPI	Enables SPI access support in the VINETIC® driver. This needs the user configuration header file (see Chapter 3.3.3.2).
-DVIN_V14_SUPPORT	Enables support of VINETIC®-4VIP/-2VIP/-0/-2CPE Version 1.4 in the VINETIC® driver.
-DVIN_V21_SUPPORT	Enables support of the VINETIC®-4M/-4C/-4S version 2.1 or greater in the VINETIC® driver.
-DVIN_DEFAULT_FW	Includes default firmware c header files at compilation time for VINETIC®-4VIP Version 1.4.
-DVINETIC_MAX_DEVICES=X	Sets the number of VINETIC® devices the VINETIC® driver should support.
-DENABLE_TRACE	Enables debugging traces at VINETIC® driver runtime.
-DRUNTIME_TRACE	Enables runtime traces of register access in VINETIC® driver. Previous option -DENABLE_TRACE must be also set.
-DENABLE_USER_CONFIG	Includes the user configuration header file in the VINETIC® driver. User defined macros will then be used instead of default macros (see Chapter 2.9).
-DTAPI_VOICE	Enable support of TAPI Voice over IP connection services in the VINETIC® driver.
-DTAPI_FAX_T38	Enables support of TAPI T38 Fax features in the VINETIC® driver.
-DTAPI_CID	Enables support of TAPI CID features in the VINETIC® driver.
-DTAPI_DTMF	Enables support of TAPI DTMF features in the VINETIC® driver.
-DTAPI_LT	Enables support of TAPI Line testing in the VINETIC® driver.
-DTAPI_GR909	Enables support of TAPI GR909 in the VINETIC® driver.

3.2 Data Types

Original data types are used for operating system specific functions and variables within the operating system adaptation files <***drv_vinetic_<os>.c***>. In any other VINETIC® driver source file, only IFX types are used as referenced in Table 5. This helps to avoid the type mismatch across several software components implemented by several developers.

*Note: These types are defined in the header file <**ifx_types.h**> which is part of the released source code.*

Table 5 VINETIC® Driver Data Types

Name	Description
IFX_char_t	Character data type.
IFX_uint8_t	Unsigned 8-bit data type.
IFX_int8_t	Signed 8-bit data type.
IFX_uint16_t	Unsigned 16-bit data type.
IFX_int16_t	Signed 16-bit data type.
IFX_uint32_t	Unsigned 32-bit data type.
IFX_int32_t	Signed 32-bit data type.
IFX_float_t	Float data type.
IFX_void_t	Void data type.
IFX_vuint8_t	Volatile unsigned 8-bit data type.
IFX_vint8_t	Volatile signed 8-bit data type.
IFX_vuint16_t	Volatile unsigned 16-bit data type.
IFX_int16_t	Volatile signed 8-bit data type.
IFX_vuint32_t	Volatile unsigned 32-bit data type.
IFX_vint32_t	Volatile signed 32-bit data type.
IFX_vfloat_t	Volatile float data type.

3.3 Relevant Driver Interfaces for the Integration

The following chapters describe the relevant interfaces used for the integration of the VINETIC® driver in a system with respect to the hardware considerations listed in **Chapter 2**.

3.3.1 VINETIC® Basic Device Initialization

The application controlled basic device initialization is the first action to be executed by the VINETIC® driver for each device available in the system. For this purpose, the VINETIC® driver has a dedicated interface called **FIO_VINETIC_BASICDEV_INIT** (see **Chapter 4.1.1**) which expects the following parameters:

1. The VINETIC® access mode, which must be set along with the system initialization (see **Chapter 4.3**),
2. The VINETIC® device physical base address, which is known by the system and applicable only for parallel access (see **Chapter 3.3.3**),
3. The VINETIC® device irq line number known by the system. If the parameter is negative, polling mode is assumed.

During the basic device initialization, following actions take place:

1. In case of parallel access, VINETIC® access addresses for NWD, EOM and DIA are set.
2. Low-level access function pointers are mapped according to the access mode (see **Chapter 4.3**).
3. When in interrupt mode, the interrupt routine is registered by the operating system.

After a successful basic initialization, following options are possible:

1. Chip access according to selected access mode (see **Chapter 3.3.3**),
2. Interrupt handling in case of interrupt mode,
3. Whole set of VINETIC® driver ioctl interfaces,
4. Complete TAPI ioctl interfaces.

3.3.2 VINETIC® Device Reset

When a single VINETIC® device is reset (see **Chapter 2.2**), the VINETIC® driver must be involved because the underlying device data must also be reset. This is done via the command **FIO_VINETIC_DEV_RESET** (see **Chapter 4.1.2**).

*Note: If a basic device initialization (see **Chapter 3.3.1**) was done previously, it will not be modified and an additional call to **FIO_VINETIC_BASICDEV_INIT** (see **Chapter 4.1.1**) is not necessary.*

Example

The following example resets the VINETIC® device number 0. Every function prefixed with <system_> must be provided by the system interface. It is assumed that the file descriptor of this VINETIC® device is available in the example.

```
/* activate reset of vinetic device 0 */
ret = system_activate_reset (0);
/* deactivate reset of vinetic device 0 */
if (ret == IFX_SUCCESS)
   ret = system_deactivate_reset (0);
/* reset internal device data in vinetic driver */
if (ret == IFX_SUCCESS)
   ret = ioctl (fdVinDev [0], FIO_VINETIC_DEV_RESET, 0);
```

3.3.3 VINETIC® Access

As described in **Chapter 2.4**, the VINETIC® can be accessed via the following interfaces:

- Parallel interface (8 bit and 16 bit): direct access, defined in drv_vinetic_parallel.c
- Serial (SPI): Access via SPI module, defined in drv_vinetic_serial.c

These interfaces assure the sequential read/write of packet and command data to and from the VINETIC® device.

The chip access low-level functions are set according to the VINETIC® access mode settings within the VINETIC® Basic Device Initialization (see **Chapter 3.3.1**).

Nevertheless, there are special considerations for the access modes described in the following chapters.

3.3.3.1 VINETIC® Parallel Access

The VINETIC® driver is provided with a set of functions pointers which are mapped according to the parallel access mode settings. Nevertheless, the support of 8-bit access needs to be enabled at compile time with the compiler switch -DVIN_8BIT. The support of the parallel access doesn't require any user specific adaptation, unless the access addresses must be interpreted differently. For this purpose, the set of shift factors defined in the user configuration file can be used (see **Chapter 2.8** and **Chapter 2.9**).

Note: The VINETIC® driver implicitly supports the 16-bit access. No compiler switch is therefore needed for this access mode.

3.3.3.2 VINETIC® SPI Access

To enable the SPI support in the VINETIC® driver, the extra compiler switch -DVIN_SPI must be used during compilation.

Note: In case of Linux®, SPI access is compiled when --enable-spi is passed as argument on the configure script or autogen.sh script

When compiled with -DVIN_SPI, the VINETIC® driver provides generic SPI low-level routines (drv_vinetic_serial.c/h) and expects some macros to be set in the user configuration header file (see **Chapter 2.9**). These macros are:

- SPI_MAXBYTES_SIZE, which indicates how many bytes can be read or written in one go via the SPI interface
- SPI_CS_SET (devnr, high_low), which sets the device SPI chip select to low or to high
- Spi_ll_read_write (txptr, txsize, rxptr, rxsize), which is mapped to the exported system low-level spi read/write routine

Once these adaptations are done, the VINETIC® SPI access will fully work with the VINETIC® driver.

Note: The SPI interface was successfully integrated and tested with the EASY 334 platform. For this platform, a PowerPC SPI driver is available and exports low-level read/write functions.

Attention: The PowerPC SPI driver must be compiled with -DNO_MUTEX_SPI to avoid use of mutexes within the exported low-level SPI functions.

3.4 VINETIC® Driver Integration

The VINETIC® driver controls the communication with the VINETIC® chip and doesn't take care of any hardware or system configuration.

Therefore, all system or platform relevant initialization and control tasks have to take place in separate software modules which must be implemented when integrating the VINETIC® driver. This makes the VINETIC® driver platform independent and reduces the porting issues on all platforms.

The following chapters describe the steps needed for the complete integration of the VINETIC® driver on a new system.

3.4.1 Integration's Big Picture

The initialization of the system (access mode, clock rate, interrupt line, chip select) must take place at system level before the VINETIC® driver is integrated. After this step, each VINETIC® device must be initialized via a provided driver interface (see **Chapter 3.3.1**) and then it must be again initialized individually i.e. with the TAPI initialization interface. **Figure 3** shows the complete integration flow:

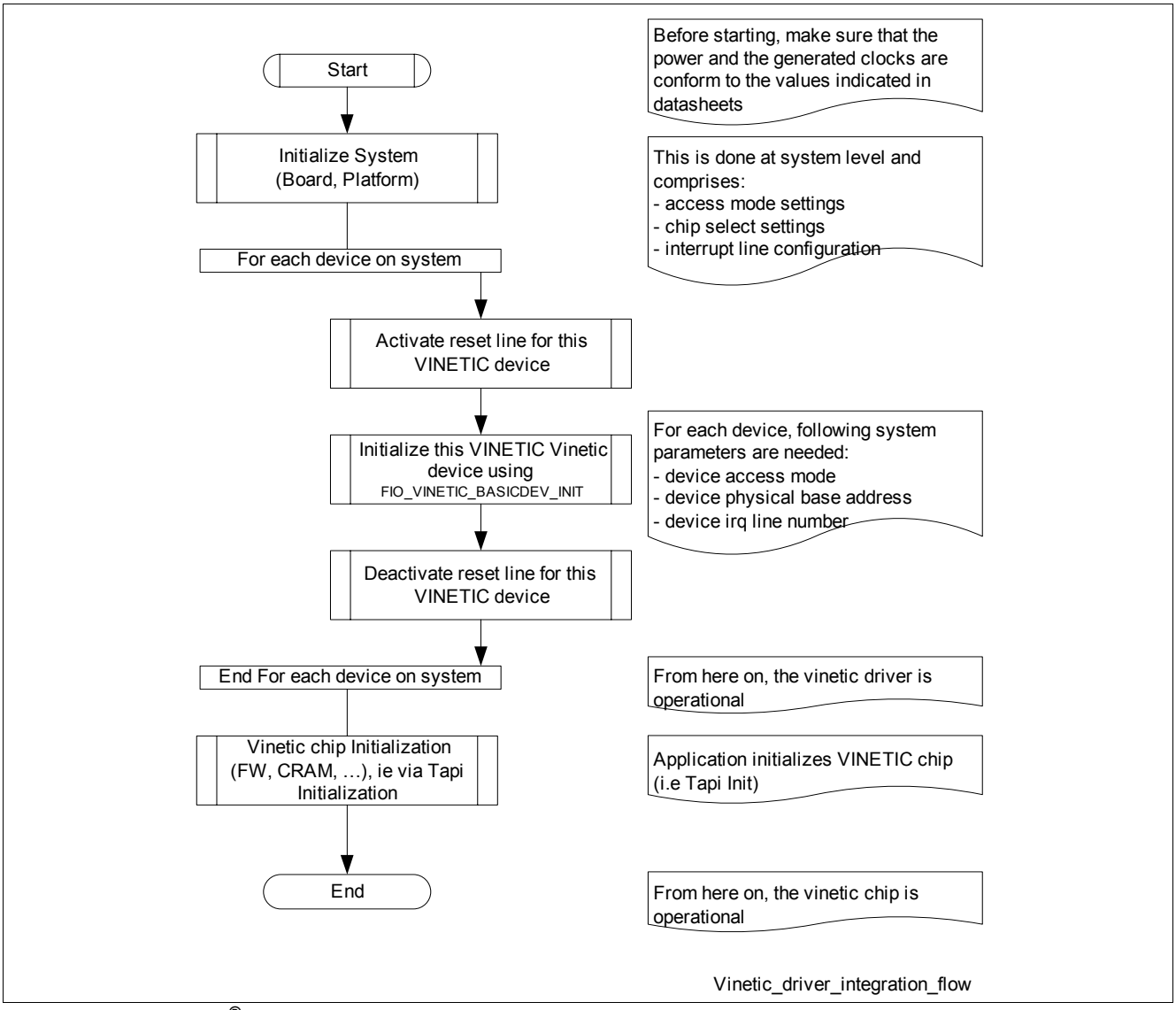

Figure 3 VINETIC® Driver Integration Flow

3.4.2 Integration Details

Software support must be provided for each hardware system integrating one or more VINETIC® devices. This support can be provided in form of a system driver (commonly called *board driver*) or in form of a BSP[1]. The following steps from 0 to 7 described below lead to a successful integration of the VINETIC® driver on a system.

Attention: In the description below, functions or macros prefixed with <system> or <SYSTEM> represent a pseudocode example of functionalities required by the system software. This does not mean that all macros/functions implemented by the system software must be prefixed as stated above. It is only important that the functionality behind the pseudocode is granted. Real life example were added for the EASY 334 system. For this system, a user library interface with an appropriate board driver has been implemented. All functions calls are from this library.

1) BSP means Board Support Package. This support can be provided as an example by or integrated to the used operating system.

3.4.2.1 Step 0

Before setting up the VINETIC® driver, it must be verified that:

- The VINETIC® device is powered accordingly.
- The clocks are set properly (see **Chapter 2.1**).

Please refer to appropriate VINETIC® data sheets (**[1]**, **[2]**, **[3]**, **[4]**, **[6]**, **[7]**, **[8]**) for details.

3.4.2.2 Step 1

Initialize the system dependent on the used VINETIC® chip by setting the access mode (see **Chapter 2.4**), the chip selects, the clock rate (see **Chapter 2.1**) and the interrupt lines (see **Chapter 2.5**).

Pseudo Code Example

```
ret = system_init (SYSTEM_16BIT_ACCESS_MODE, SYSTEM_2048KHZ_CLOCKRATE);
```

Note: This is an example. May be implemented differently in user's system software.

Example

```
ret = easy334_init (EASY334_16BIT_ACCESS_MODE, EASY334_2048KHZ_CLOCKRATE);
```

3.4.2.3 Step 2

Activate the reset line for each VINETIC® device (see **Chapter 2.2**).

Pseudo Code Example

```
ret = system_activate_reset (SYSTEM_VINETIC_DEVICE_ONE);
```

Note: This is an example. May be implemented differently in user's system software.

Example

```
/* EASY334 has only one vinetic device */
ret = easy334_activate_reset (0);
```

3.4.2.4 Step 3

Do basic device driver initialization of each VINETIC® device (see **Chapter 3.3.1**).

Attention: It is assumed that the VINETIC® driver is already installed and that all VINETIC® devices file descriptors are available.

Example

```
VINETIC_BasicDeviceInit_t devInit;

memset (&devInit, 0, sizeof(devInit));
/* set access mode according to VIN_ACCESS enumeration */
devInit.AccessMode = VIN_ACCESS_PAR_16BIT;
devInit.nBaseAddress = 0xC0010000;
devInit.nIrqNum = 12;
ret = ioctl (fdVinDev, FIO_VINETIC_BASICDEV_INIT, &devInit);
```

Note: This implementation can be used as generic code to basically initialize each VINETIC® device. Values are examples.

3.4.2.5 Step 4

Deactivate the reset line for the initialized VINETIC® device (see **Chapter 2.2**).

Pseudo Code Example

```
ret = system_deactivate_reset (SYSTEM_VINETIC_DEVICE_ONE);
```

Note: This is an example. May be implemented differently in user's system software.

Example

```
ret = easy334_deactivate_reset (0);
```

3.4.2.6 Step 5

Read the version of the VINETIC® device as an access test.

Generic Code Example

```
VINETIC_IO_VERSION devVers;

memset (&devVers, 0, sizeof(devVers));
ret = ioctl (fdVinDev, FIO_VINETIC_VERS, &devVers);
if (ret == IFX_SUCCESS)
   printf ("VINETIC [version 0x%2X, type 0x%2X, channels %d] ready!\n\r",
                    devVers.nChip, devVers.nType, devVers.nChannel);
```

Note: This implementation can be used as generic code to read the version of each VINETIC® device.

3.4.2.7 Step 6

Execute TAPI initialization (Firmware download/activation, CRAM download etc....) and feed the analog channels lines for each channel on the VINETIC® device.

Example

```
TAPI_INIT tapiInit;

memset (&tapiInit, 0, sizeof(tapiInit));
tapiInit.nMode = TAPI_VOICE_CODER;
ret = ioctl (fdVinChan, IFXPHONE_INIT, &tapiInit);
if (ret == SUCCESS)
   ret = ioctl (fdVinChan, IFXPHONE_SET_LINEFEED, 0);
```

3.4.2.8 Step 7

Driver interface is operative with VINETIC® using appropriate functions (read/write, TAPI ioctls).

3.4.3 Advanced Integration Code Example

The following code example is a copy/paste integration code supposed to work on your platform with only small modifications. It is a pseudo application code to bring up the VINETIC® driver in your system. It is assumed that all VINETIC® devices file descriptors are available in the example.

Attention: Infineon Technologies can not guarantee that this code will work, as it is dependent on the system. The order of the calls must not be changed.

Example

```
IFX_int32_t ret, nDevNum, nAccessMode, i;
VINETIC_BasicDeviceInit_t devInit;
VINETIC_IO_VERSION devVers;

memset (&devInit, 0, sizeof(devInit));
memset (&devVers, 0, sizeof(devVers));
/* initialize the system and get back number of vinetic devices.
   During this initialization, following will be initialized:
   - access mode
   - clock rate
   - chip select(s)
   - interrupt line
*/
ret = system_init (&nDevNum);
if (ret == IFX_ERROR)
{
   printf ("system initialization fails\n\r");
   return ret;
}

/* in case of successful init, for all devices in the system:
   - read basic parameters from system interface
   - activate vinetic device reset line
   - do vinetic basic device initialization
   - deactivate vinetic device reset
   - read vinetic device version
*/
for (i = 0; i < nDevNum; i++)
{
   /* get basic parameters: access mode, base address, irq number */
   ret = system_get_parameter (i, &(devInit.AccessMode),
                               &(devInit.nBaseAddress), &(devInit.nIrqNum));
   if (ret == IFX_ERROR)
      break;
   /* activate vinetic device reset line */
   ret = system_activate_reset (i);
   if (ret == IFX_ERROR)
      break;
   /* do basic device initialization */
   ret = ioctl (fdVinDev[i], FIO_VINETIC_BASICDEV_INIT, &devInit);
   if (ret == IFX_ERROR)
      break;
   /* deactivate vinetic device reset line now */
   ret = system_deactivate_reset (i);
   if (ret == IFX_ERROR)
      break;
   /* read vinetic device version as test */
   ret = ioctl (fdVinDev[i], FIO_VINETIC_VERS, &devVers);
   if (ret == IFX_ERROR)
      break;
```

```
   /* print out version */
   printf ("VINETIC [version 0x%2X, type 0x%2X, channels %d] ready!\n\r",
                    devVers.nChip, devVers.nType, devVers.nChannel);
}

if (ret == IFX_ERROR)
   printf ("system integration fails in regard to vinetic\n\r");

return ret;
```

3.5 Line Testing Integration

Floating point operations can not be done on the kernel level (where the driver runs) for some operating systems. For line testing measurements therefore the driver passes the results to an application level software for calculations. For this purpose, a TAPI line testing library is available. This library currently supports **Linux®** and **VxWorks®** operating systems. In case the line testing support is needed for another operating system, this library must be adapted (refer to **[19]** for more details).

4 Quick Reference

The references listed here are just related to the integration issue and offer a quick overview. For VINETIC® driver references, include the file “vinetic_io.h“ (refer to **[17]** for more details).

4.1 I/O Control Reference

Lists I/O control related to the device initialization and reset.

Table 6 I/O Control Defines

Name	Description
FIO_VINETIC_BASICDEV_INIT	Does the VINETIC® basic device initialization.
FIO_VINETIC_DEV_RESET	Resets the device internal data structure.

4.1.1 FIO_VINETIC_BASICDEV_INIT

This interface must be called at first to initialize the VINETIC® driver with the chip parameters passed as pointer to the structure VINETIC_BasicDeviceInit_t. No chip access will be done until this basic initialization is successful.

```
#define FIO_VINETIC_BASICDEV_INIT _IO(VINETIC_IOC_MAGIC, 200)
```

Data type	Name	Description
Int	fd	Pointer to a file descriptor
Int	FIO_VINETIC_BASICDEV_INIT	I/O control identifier for this operation
Int	param	Use structure VINETIC_BasicDeviceInit_t

4.1.2 FIO_VINETIC_DEV_RESET

Resets internal data of the device.

```
#define FIO_VINETIC_DEV_RESET _IO(VINETIC_IOC_MAGIC, 201)
```

Note: Neither the (physical) reset pin/signal of the VINETIC®, nor the internal state of the EDSP will be affected by this IOCTL.

Data type	Name	Description
Int	fd	Pointer to a file descriptor
Int	FIO_VINETIC_DEV_RESET	I/O control identifier for this operation
Int	param	0

4.2 Structures

This chapter addresses structure type definition issues.

Table 7 Structures Used

Name	Description
VINETIC_BasicDeviceInit_t	VINETIC® basic device initialization structure.

4.2.1 VINETIC_BasicDeviceInit_t

Basic device initialization structure.

```
typedef struct {
  VIN_ACCESS AccessMode;
```

```
  unsigned long nBaseAddress;
  signed int nIrqNum,
}VINETIC_BasicDeviceInit_t;
```

Data type	Name	Description
VIN_ACCESS (see **Table 9**)	AccessMode	Access mode for VINETIC® device, 8 bit or 16 bit or serial (SPI)
Unsigned long	nBaseAddress	VINETIC® physical base address
Signed int	nIrqNum	VINETIC® device irq number, as defined by the OS. If the irq number is set to -1, the driver will be configured in polling mode.

4.3 Enumerator Reference

This chapter addresses the enumerator reference.

Table 8 Enumerator reference

Name	Description
VIN_ACCESS (see **Table 9**)	VINETIC® access modes

VINETIC® access modes (see **Chapter 2.4**):

Table 9 VIN_ACCESS

Name	Value	Description
VIN_ACCESS_SPI	0	SPI access, V**1**.x/**2**.x.
VIN_ACCESS_SCI	0x01	SCI access, same as SPI access, V**1**.x /**2**.x.
VIN_ACCESS_PAR_16BIT	0x2	16-bit Motorola parallel access, V**1**.x/**2**.x.
VIN_ACCESS_PAR_8BIT	0x3	8-bit Motorola parallel access, V**1**.x/**2**.x.
VIN_ACCESS_PARINTEL_MUX16	0x4	16-bit Intel multiplexed parallel access, V**2**.x only
VIN_ACCESS_PARINTEL_MUX8	0x5	8-bit Intel multiplexed parallel access, V**2**.x only
VIN_ACCESS_PARINTEL_DMUX8_BE	0x6	8-bit Intel demultiplexed access, big endian, V**1**.x only
VIN_ACCESS_PARINTEL_DMUX8_LE	0x7	8-bit Intel demultiplexed access, little endian, V**2**.x only
VIN_ACCESS_PAR_8BIT_V2	0x8	8-bit parallel motorola access for VINETIC® V**2**.x. only

References

[1] VINETIC®-4VIP (PEB 3324) Version 1.4 Prel. Data Sheet DS1, 2003-08-07

[2] VINETIC®-2VIP (PEB 3322) Version 1.4 Prel. Data Sheet DS1, 2003-08-07

[3] VINETIC®-2CPE (PEB 3332) Version 1.4 Prel. Data Sheet Rev. 3.0, 2005-07-18

[4] VINETIC®-0 (PEB 3320) Version 1.4 Prel. Data Sheet DS1, 2003-08-07

[5] Prel. Addendum Rev. 2.0 to VINETIC® Version 1.4 Prel. Data Sheet DS1, 2004-08-16

[6] VINETIC®-4M (PEF 3314) Version 2.1/Version 2.2 Prel. Data Sheet Rev. 2.0, 2004-11-10

[7] VINETIC®-4C (PEF 3394) Version 2.1/Version 2.2 Prel. Data Sheet Rev. 2.0, 2004-11-10

[8] VINETIC®-4S (PEF 3304) Version 2.1 Prel. Data Sheet Rev. 2.0, 2005-04-12

[9] VINETIC® Version 1.4 Prel. User's Manual – Software Description Rev. 2.0, 2004-10-19

[10] VINETIC®-4M/-4C (PEF 3314/-3394) Version 2.1/Version 2.2 Prel. User's Manual – Software Description Rev. 2.0, 2005-01-20

[11] Prel. Addendum Rev. 1.0 to VINETIC®-4M/-4C (PEF 3314/-3394) Version 2.1/Version 2.2 Prel. User's Manual – Software Description Rev. 2.0, 2005-06-30

[12] VINETIC®-4S (PEF 3304) Version 2.1 Prel. User's Manual – Software Description Rev. 1.0, 2005-03-22

[13] Prel. Addendum Rev. 1.0 to VINETIC®-4S (PEF 3304) Version 2.1 Prel. User´s Manual – Software Description Rev. 1.0, 2005-06-30

[14] VINETIC® Version 1.4/2.1 Prel. User's Manual – EDSP Firmware Description Rev. 1.0, 2004-06-18

[15] VINETIC® Version 1.4/2.1/2.2 Prel. User's Manual – EDSP Firmware Description Rev. 2.0, in preparation

[16] VINETIC®-4M/-4C (PEF 3314/-3394) Version 2.2 EDSP Prel. Firmware Overview Rev. 2.0, 2005-06-14

[17] VINETIC® Prel. User's Manual Software Description - Driver Rev. 4.0, 2004-12-08

[18] VINETIC® Version 1.3/Version 1.4 Prel. Status Sheet PHI Download DS2, 2003-07-02

[19] VINETIC® (PEB/PEF 33xy) Version 1.4 and 2.1/2.2 Application Note Telephony API (TAPI) V2.3 Rev. 16, 2005-04-13

[20] VINETIC® Version 2.1/2.2 Preliminary Hardware Design Guide Rev. 2.0, 2005-05-19

[21] VINETIC® Version 2.1/2.2 Prel. Errata Sheets

[22] VINETIC® Version 1.4 Prel. Errata Sheets

[23] VINETIC® EDSP Firmware Release Notes

[24] VINETIC® Driver Software V0.11.8 Release Note Rev. 1.0

[25] VINETIC® Driver Software V1.0 Release Note Rev. 1.0

www.infineon.com

Published by Infineon Technologies AG