Application Interrupt Control with VStdLib

Version 1.0
2008-08-06
Application Note AN-ISC-2-1081

Author(s)	Patrick Markl
Restrictions	Restricted Membership

Abstract — This application note explains how the application can control interrupt handling via the VStdLib and which constraints apply.

Table of Contents

1.0 Overview

This application note describes how the user can configure the interrupt control options of the VStdLib. Some applications provide their own lock/unlock functions, which better fulfill the application’s needs. Because of this the VStdLib provides a means which allows the application to use it’s own lock/unlock functions, instead of the implementation provided by the VStdLib.

This application note describes the handling of this use case in more detail.

1.1 Introduction

The VStdLib provides functions to lock/unlock interrupts. There are three options to be set in the configuration tool, as shown in figure1. The first option (Default) lets the VStdLib lock global interrupts. Depending on the hardware plattform it is also possible to lock interrupts to a certain level. The lock is implemented by the VStdLib itself.

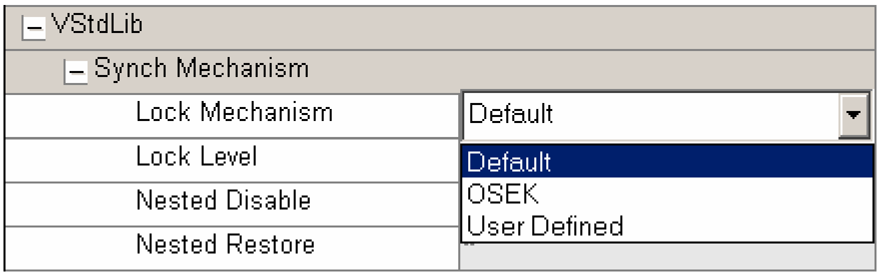
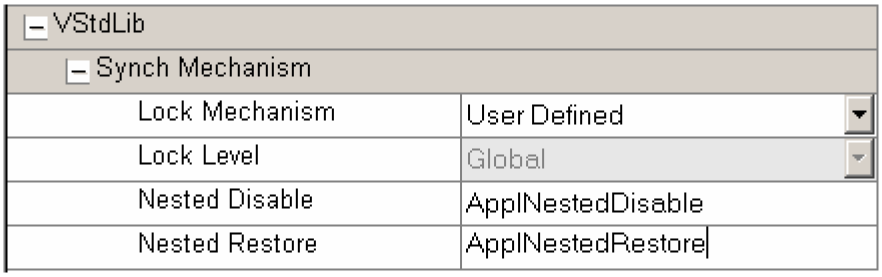


Figure 1: Possible configuration options for VStdLib interrupt control

The second option (OSEK) is to configure the VStdLib in a way that locking of interrupts is done by means of OSEK OS functions. The third and last option (User defined) requires the application to perform the locking/unlocking functionality within callback functions.

This application note focusses mainly on the third option. It describes the way the application has to implement the callback functions required by the VStdLib.



VStdLib
Synch Mechanism

Lock Mechanism	User Defined
Lock Level	Global
Nested Disable	ApplNestedDisable
Nested Restore	ApplNestedRestore

Figure 2: Configuration of interrupt control by application

Figure 2 shows the VStdLib configuration dialog, if interrupt control by application is configured. The user has to enter the names of two functions in the dialog, which will be called by the VStdLib in order to lock/unlock interrupts. If the user has specified the callback function names as shown in figure 2, the application must provide the implementations of these two two functions. The prototypes are:

```
void ApplNestedDisable(void);

void ApplNestedRestore(void);
```

From now on these two function names will be used within this application note.

These two functions are called by the VStdLib, in case any Vector component requests a lock for a critical section. The user has to make sure that the locking mechanism within these two functions is sufficient to protect data. This depends heavily on the architecture of the application. The more priority levels exists, which call Vector functions, the more restrictive the lock must be.

⚠	Please check the technical references of the other Vector components for restrictions regarding the call context of the API functions.

2.0 Interrupt Control by Application

This configuration option is usually used, if a global lock is not desired by the user or special lock mechanisms are used. Once this option is configured, there are two functions to be provided by the application. The user can specify the names of these functions in the configuration dialog of the VStdLib. The VStdLib calls these functions instead of directly locking/unlocking interrupts. This means, if any Vector component requests an interrupt lock, it is finally performed by the application provided functions.

The first function is called, in order to perform a lock operation. It is expected, that the application function stores the current interrupt state(or any other), in order to restore it later. The second function is to restore the previously saved lock state.

The implementation of these two functions is up to the user. The user may lock just certain interrupt sources or set a mutex, semaphore or whatever ensures consistent data and fulfills the call context requirements, described in the Vector component specific technical references.

2.1 Constraints

The usage of Interrupt Control by Application has some constraints, which have to be taken into account. The following chapters describe them.

2.1.1 Constraint 1: Nested Calls

It is expected that the two callback functions (ApplNestedDisable()/-Restore()) are implemented in a way that nested calls are possible. This means if the function ApplNestedDisable() was called by some software component it may happen that this function is called again from somewhere else. This has to be taken into account when saving and restoring the interrupt state! The implementer of these two function can assume that the number of lock and unlock calls is identical and nesting is balanced.

2.1.2 Constraint 2: Recursive Calls when Disabling CAN Interrupts

Instead of implementing an own lock mechanism, the user could configure interrupt control by application and call the CAN driver's CanCanInterruptDisable()/-Restore() functions. These two function simply disable CAN interrupts for the given CAN channel. These two CAN driver functions protect the access to their state variables by means of the VStdLib's lock mechanism, which would again be implemented by the callbacks provided by the application. This would cause an indirect recursion.

⚠	Please note that CanCanInterruptDisable()/Restore() shall not be called from ApplNestedDisable()/Restore(). This application note does not provide a solution for this use case!

2.1.3 Constraint 3: No Locking when Disabling CAN Interrupts

One could think of letting the application directly modify the interrupt flags of the CAN controller, to overcome the recursion, described in the previous chapter. But this would cause the CAN interrupts to be never locked, when CanCanInterruptDisable() is called, by any component. The reason is that the application's interrupt lock code would interfere with the code in the CAN driver's function CanCanInterruptDisable(). The following pseudo code shows the way CanCanInterruptDisable() is implemented. It is assumed that ApplNestedDisable()/-Restore() are implemented to allow nested calls.

```
/* CAN Interrupt will be never locked in this example!!! */
void CanCanInterruptDisable(CAN_CHANNEL_CANTYPE_ONLY)
{
  ApplNestedDisable();
  Lock CAN interrupts
  ApplNestedRestore();
}

void ApplNestedDisable(void)
{
  Save current CAN interrupt state();
  Lock CAN Interrupts();
}

void ApplNestedRestore(void)
{
  Restore CAN interrupts to previous state();
}
```

Figure 3 shows what happens in this case. The function CanCanInterruptDisable() calls ApplNestedDisable() in order to protect an internal counter. This lock function disables the CAN interrupts, afterwards the CAN driver's function locks the CAN interrupts too. The next thing is to call ApplNestedRestore() which again is implemented by the application and restores the previous CAN interrupt state – in this case enables the CAN interrupts. Now an inconsistency exists. The code which called CanCanInterruptDisable() assumes locked CAN interrupts, but they aren't

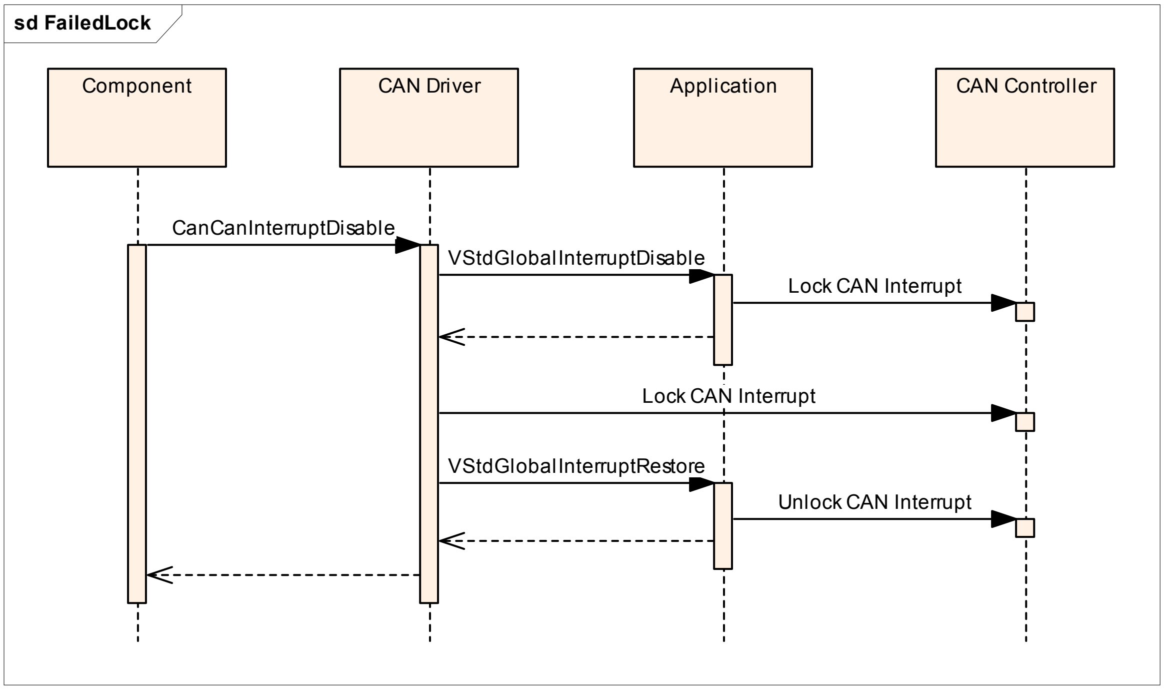


Figure 3: Sequence diagram of CanCanInterruptDisable()

3.0 Solution

This chapter proposes a solution and a code examples, to overcome the constraints 1 and 3 described in the previous chapters.

3.1.1 Nested Calls

Solving the first issue – nested calls – is simply done by introducation of a nesting counter. The callbacks implemented by the application need to manage this counter. The counter is to be incremented, if the function to lock interrupts is called and decremented if the unlock function is called. The application has to take care to initialize this counter, before any Vector function is called, in order to ensure a consistent interrupt locking.

The interrupt state is to be modified only if the counter has the value zero. If the value is greater than zero, the counter is just maintained. The following code example shows, how this nested counter could be implemented.

```
/* Global variable as nesting counter */
vuint8 gApplNestingCounter;

/* Must be called before the Vector components are initialized! */
void SomeApplicationInitFunction(void)
{
  gApplNestingCounter = (vuint8)0;
}

void ApplNestedDisable(void)
{
  /* check counter - lock if counter is 0 */
  if((vuint8)0 == gApplNestingCounter)
  {
    /* Save current state and perform lock  */
    ApplicationSpecificSaveStateAndLock();
  }
  /* increment counter - do not disable if nested, because already done */
  gApplNestingCounter++;
}

void ApplNestedRestore(void)
{
  gApplNestingCounter--;
  if((vuint8)0 == gApplNestingCounter)
  {
    ApplicationSpecificRestoreToPreviousState();
  }
}
```

3.1.2 No Locking of Interrupts

Constraint 3 described a situation, in which the CAN interrupts are not locked at all. This is because the application implements a lock function, which modifes the CAN interrupt registers with own code. To overcome this issue, a global flag needs to be implemented. This global flag tells the application, when to lock or unlock CAN interrupts. The flag is set within two additional callback functions to be implemented by the application. The prototypes of the additional callbacks are:

void ApplCanAddCanInterruptDisable(CanChannelHandle channel);

void ApplCanAddCanInterruptRestore(CanChannelHandle channel);

The callback functions are called by the CAN driver from within the functions CanCanInterruptDisable() and CanCanInterruptRestore() and have to be enabled by means of the preprocessor define C_ENABLE_INTCTRL_ADD_CAN_FCT. This is done, by creating a user config file, which contains this definition. More information about these two functions can be found in [1].

The sequence diagrams in figure 4 and figure 5 show the lock and unlocking procedure respectively.

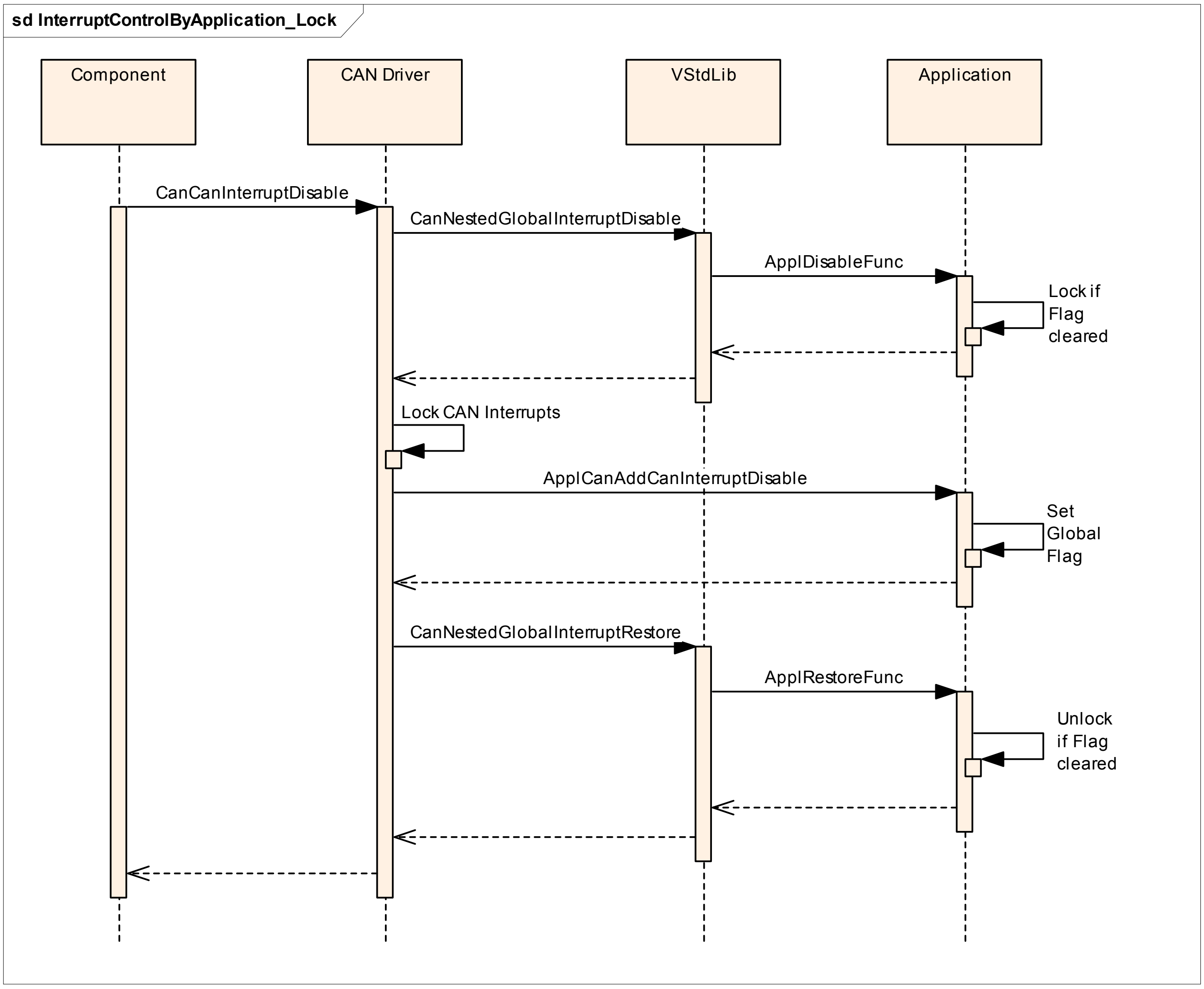


Figure 4: Sequence diagram for locking just CAN interrupts.

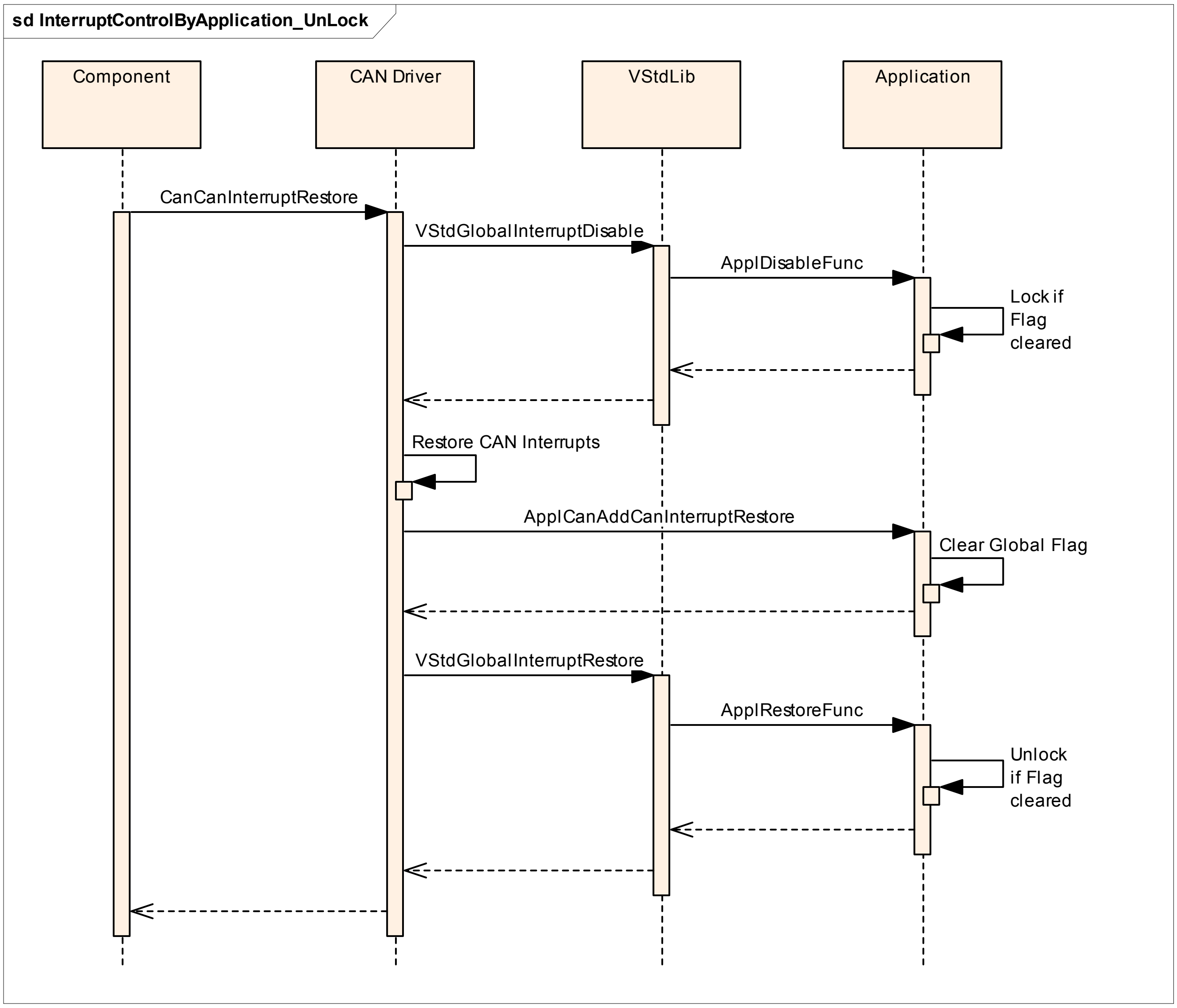


Figure 5: Sequence diagram for unlocking just CAN interrupts

The following code example shows how to implement the handling of the global flag. If the function CanCanInterruptDisable() is called, it calls the ApplNestedDisable(), in order to protect a counter. This function locks CAN interrupts using own code. When ApplNestedDisable() returns, the CAN driver locks CAN interrupts too. Afterwards ApplCanAddCanInterruptDisable() is called. This function is implemented by the application and sets the global flag. Before the function CanCanInterruptDisable() returns, it calls ApplNestedRestore(). The application, which implements the restore callback function has to check, if the global flag is set. If yes, the CAN interrupts must not be unlocked!

If the function CanCanInterruptRestore() is called, first ApplNestedDisable() is called again. Then the CAN driver unlocks the CAN interrupts (if its nesting counter reached the value zero) and calls the function ApplCanAddCanInterruptRestore(). Within this function the flag is cleared. If ApplNestedRestore() is called now, the flag is not set anymore and the restore of the CAN interrupts is performed.

Note that the application needs to implement also a nesting counter, if it uses own code to lock CAN interrupts, in order to avoid the issue described by constraint 1. The following code example shows, how to implement the nesting counter and the flag.

```
vuint8 gCanLockFlag;
vuint8 gApplNestingCounter;

void ApplicationInitFunction(void)
{
  /* initialize the flags */
  gCanLockFlag = (vuint8)0;
  gApplNestingCounter = (vuint8)0;
}

void ApplNestedDisable(void)
{
  if((vuint8)0 == gApplNestingCounter)
  {
    if((vuint8)0 == gCanLockFlag)
    {
      Save current CAN interrupt state();
      Lock CAN Interrupts();
    }
  }
  gApplNestingCounter++;
}

void ApplNestedRestore (void)
{
  gApplNestingCounter--;
  if((vuint8)0 == gApplNestingCounter)
  {
    if((vuint8)0 == gCanLockFlag)
    {
      Restore CAN interrupts to previous state();
    }
  }
}

void ApplCanAddCanInterruptDisable(CanChannelHandle channel)
{
  gCanLockFlag = (vuint8)1;
}

void ApplCanAddCanInterruptRestore(CanChannelHandle channel)
{
  gCanLockFlag = (vuint8)0;
}
```

4.0 Referenced Documents

The following table contains the referenced documents.

Referenced Documents
[1] TechnicalReference_CANDriver.pdf

5.0 Contacts

Vector Informatik GmbH Ingersheimer Straße 24 70499 Stuttgart Germany Tel.: +49 711-80670-0 Fax: +49 711-80670-111 Email: info@vector-informatik.de	**Vector CANtech, Inc.** 39500 Orchard Hill Pl., Ste 550 Novi, MI 48375 USA Tel: +1-248-449-9290 Fax: +1-248-449-9704 Email: info@vector-cantech.com	**VecScan AB** Lindholmspiren 5 402 78 Göteborg Sweden Tel: +46 (0)31 764 76 00 Fax: +46 (0)31 764 76 19 Email: info@vecscan.com
Vector France SAS 168 Boulevard Camélinat 92240 Malakoff France Tel: +33 (0)1 42 31 40 00 Fax: +33 (0)1 42 31 40 09 Email: information@vector-france.fr	**Vector Japan Co. Ltd.** Seafort Square Center Bld. 18F 2-3-12, Higashi-shinagawa, Shinagawa-ku J-140-0002 Tokyo Tel.: +81 3 5769 6970 Fax: +81 3 5769 6975 Email: info@vector-japan.co.jp	**Vector Korea IT Inc.** Daerung Post Tower III, 508 Guro-gu, Guro-dong, 182-4 Seoul, 152-790 Republic of Korea Tel.: +82-2-2028-0600 Fax: +82-2-2028-0604 Email: info@vector-korea.com