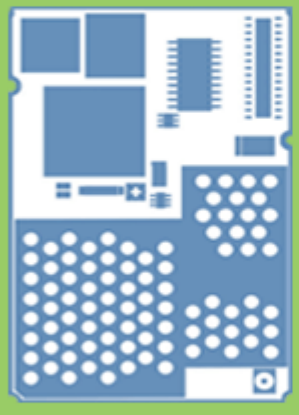
CINTERION
a Gemalto company

Java User's Guide

Version: 19
DocId: wm_java_usersguide_v19

Products: TC65i, TC65i-X, EGS5, EGS5-X



User's Guide

Document Name:	**Java User's Guide**
Version:	**19**
Date:	**2012-01-27**
DocId:	**wm_java_usersguide_v19**
Status	**Confidential / Released**
Supported Products:	**TC65i, TC65i-X, EGS5, EGS5-X**

Content

Tables

Figures

1 Preface

This document covers the full range of IMP-NG Java products from Cinterion Wireless Modules, currently including:

1. TC65i Module
2. TC65i-X Module
3. EGS5 Module
4. EGS5-X Module

Differences between the products are noted in the particular chapters. Throughout the document, all supported products are referred to as ME (Mobile Equipment). For use in file, directory or path names, the string "<productname>" represents the actual name of a product, for example TC65i. Screenshots are provided as examples and, unless otherwise stated, apply to all supported products.

2 Overview

The ME features an ultra-low profile and low-power consumption for data (CSD and GPRS), voice, SMS and fax. Java technology and several peripheral interfaces on the module allow you to easily integrate your application.

This document explains how to work with the ME, the installation CD and the tools provided on the installation CD.

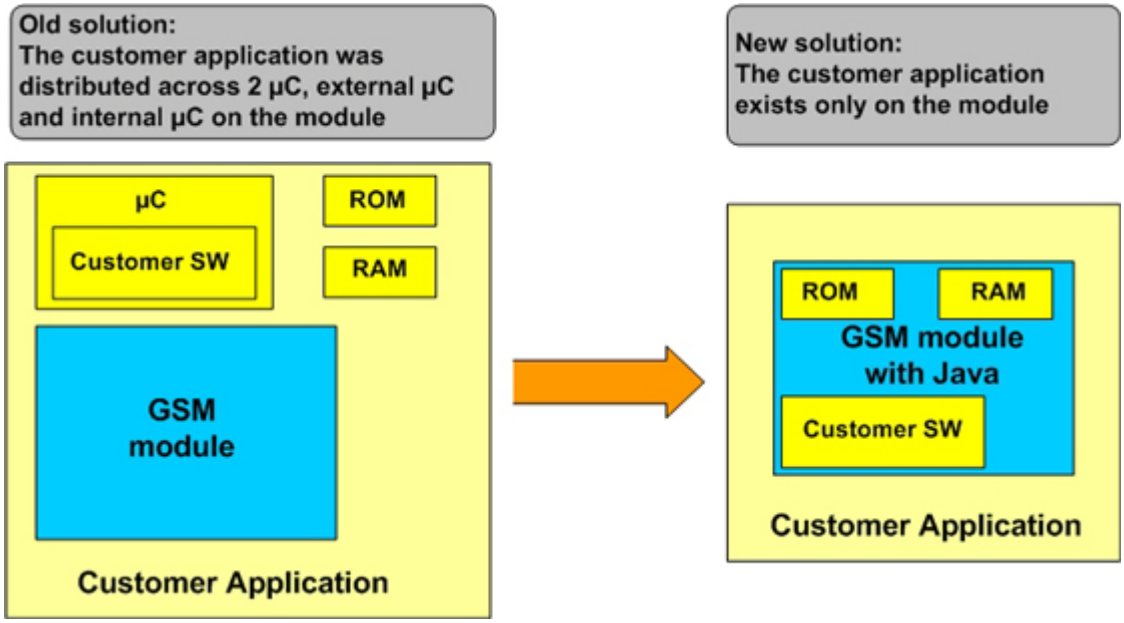


Figure 1: Overview

2.1 Related Documents

List of documents referenced throughout this manual:
[1] AT Command Set of your Cinterion Wireless Modules product
[2] Hardware Interface Description of your Cinterion Wireless Modules product
[3] Java doc \wtk\doc\html\index.html
[4] IMP-NG, JSR228, Standard

2.2 Terms and Abbreviations

Abbreviation	Description
API	Application Program Interface
ASC	Asynchronous Serial Controller
CLDC	Connected Limited Device Configuration
CMTK	Cinterion Mobility Toolkit
CSD	Circuit-Switched Data
DAI	Digital Audio Interface
DCD	Data Carrier Detect
DSR	Data Set Ready
GPIO	General Purpose I/O
GPRS	General Packet Radio Service
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
IP	Internet Protocol
Java ME™	Java Micro Edition (also known as J2ME)
Java SE™	Java Standard Edition
JAD	Java Application Description
JAR	Java Archive
JDK	Java Development Kit
JVM	Java Virtual Machine
LED	Light Emitting Diode
ME	Mobile Equipment
MES	Module Exchange Suite
MIDP	Mobile Information Device Protocol
OTA	Over The Air
OTAP	Over The Air Provisioning of Java Applications
PDP	Packet Data Protocol
PDU	Protocol Data Unit
SDK	Standard Development Kit
SMS	Short Message Service
TCP	Transfer Control Protocol
URC	Unsolicited Result Code
URL	Universal Resource Locator
VBS	Visual Basic Script
WTK	Wireless Toolkit

3 Installation

3.1 System Requirements

The Cinterion Mobility Toolkit (CMTK) requires that you have:
- Windows XP, Windows Vista or Windows 7 installed
- 110 Mbytes free disk space for the CMTK (without JDK and IDE)
- Administration privileges
- Java SE Development Kit 6 Update 25. To install the JDK version 1.6.0_25 provided on CD, follow the instructions in Section 3.3.1.

If a Java IDE such as NetBeans (as of 6.5 - 6.9.1) or Eclipse (as of 3.6.0 - Helios SR2) is installed, the CMTK environment can be integrated into it during installation of the CMTK. To install one of the IDEs, follow the installation instructions in Section 3.3.1.

If you wish to access the module via USB ensure that the USB cable is plugged between the module's USB interface and the PC.

3.2 Installation CD Content

The Cinterion Mobility Toolkit Installation CD includes:
- Wireless Toolkit. The WTK is the directory where all the necessary components for product specific Java application creation and debugging are stored. The WTK version is stored in a text file under "Program Files\Cinterion\CMTK\<product name>\WTK\VersionWTK.txt". The WTK is distributed on the CD under "program files\Cinterion\CMTK\ABC2\WTK" with
  - bin
    various tools
  - doc
    html
  - lib
    classes.zip
- WTK samples are distributed on the CD under "All Users\Cinterion ABC2 WTK Examples"
- Module Exchange Suite. The MES setup is distributed on CD under "Installer\MES-Setup.exe". The MES provides tools to access the Flash file system on the module from the development environment over a serial interface. The MES may be installed separately, but can also be installed as part of CMTK - see Section 3.3.
- IMP Debug Connection. The setup is found under "Installer\IMPDbgConnectionSetup.exe". The setup installs an IDE debug modem for on device debugging (see Chapter 10). The IMP Debug Connection may be installed separately, but can also be installed as part of CMTK - see Section 3.3.
- Java SDK
  - jdk-6u25-windows-i586.exe
- NetBeans IDE 6.9.1
  - netbeans-6.9.1-ml-javase-windows.exe
- Eclipse Helios SR2 (v3.6.1)
  - eclipse-pulsar-helios-SR2-win32.zip
- Eclipse Helios SR2 (64-bit) (v3.6.1)
  - eclipse-pulsar-helios-SR2-win32-x86_64.zip
- Documentation is distributed on the CD under "program files\Cinterion\CMTK\ABC2\<product name> Documentation" and includes AT Command Set, Hardware Interface Description as well as this Java Users Guide.

3.3 Cinterion Mobility Toolkit Installation

The CMTK is distributed on CD. The installation program automatically installs the necessary components and IDE integrations. The software can be uninstalled and updated with the install program.

This section covers the installation and removal of the CMTK including the installation of the prerequisite JDK and supported IDEs.

3.3.1 Installation Prerequisites

Before the CMTK is installed from CD a standard Java Development Kit (JDK) should be installed

- The Java SE Development Kit 6 Update 25 is distributed as part of the installation CD under "Contribution\jdk-6u25-windows-i586.exe". To install the JDK please call the contribution file and follow the instructions on the screen. If there is no appropriate JDK installed, the installation of the provided JDK will be offered automatically during the CMTK installation process. Once the JDK has been installed, the environment variable "path" can be altered to comfortably use the JDK tools without IDE (this is not necessary for using the Cinterion CMTK):
  - Open the Control Panel:
    Open System.
    Click on Advanced.
    Click on the Environment Variables button.
    Choose path from the list of system variables.
    Append the path for the bin directory of the newly installed SDK to the list of directories for the path variable.

Apart from the JDK installation it is recommended to install a Java Development Environment (IDE):

- The Eclipse IDE as well as the NetBeans IDE are distributed as part of the installation CD under "Contribution\eclipse-pulsar-helios-SR2-win32.zip" resp. "Contribution\eclipse-pulsar-helios-SR2-win32-x86_64.zip" (for 64-bit systems) or "Contibution\netbeans-6.9.1-ml-javase-windows.exe". To install any of these IDEs please call the contribution setup file (resp. unzip the contribution archive and call the setup file). An Eclipse IDE may also be installed as part of the CMTK installation described in Section 3.3.2.
  Note that the Eclipse IDE provided on the installation CD is a special Eclipse Mobile Tools Platform called Pulsar. Pulsar already includes "Mobile Tools for Java" required for developing Java applications for Cinterion Java modules. If employing any other Eclipse IDE variant or the NetBeans IDE also provided on the installation CD, it is necessary to install additional plugins containing mobile resp. mobility tools. Please refer to Chapter 10 for more information on how to install these plugins for Eclipse and NetBeans.

3.3.2 Installing CMTK

Before you start the installation please make sure all applications - especially possible IDEs - are closed:

1. Insert the installation CD and start Setup.exe. When the dialog box appears press the Next button to start the CMTK installation.

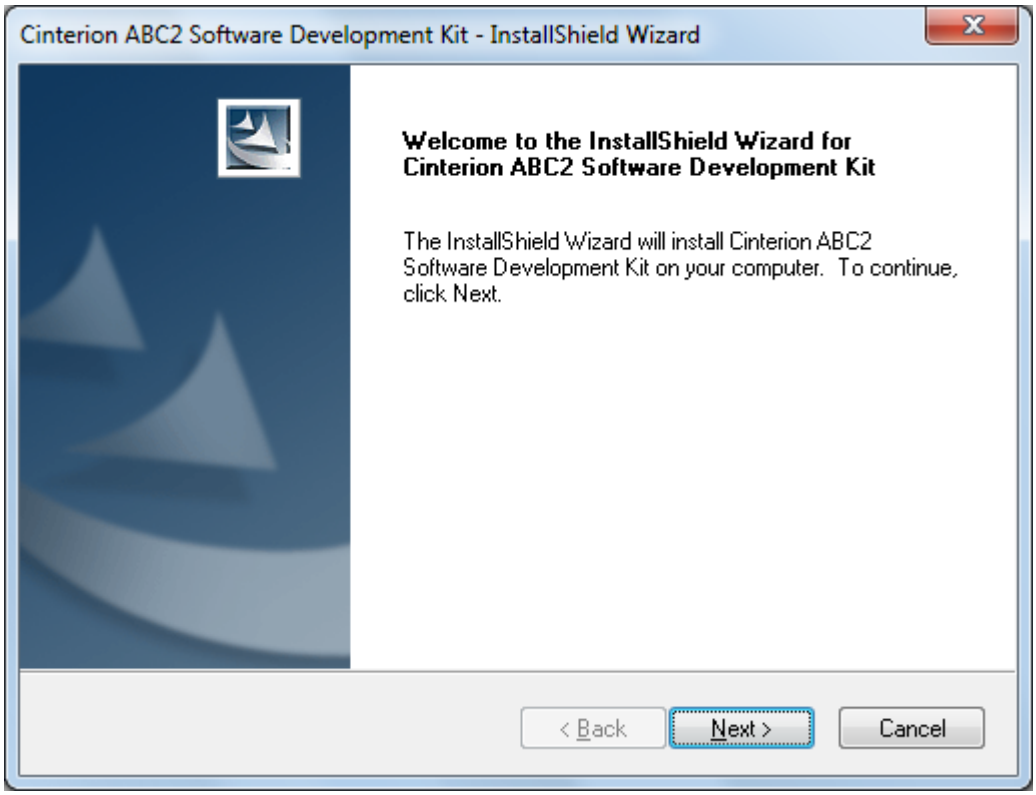


Figure 2: CMTK - InstallShield Wizard

2. Read the CMTK license agreement. If you accept the agreement, press Yes to continue with the installation.
3. Read the information about the installation and the use of the CMTK. Press Next to continue.

4. Install the Module Exchange Suite (MES) as part of the CMTK installation. The MES provides tools to access the Flash file system on the module from the development environment over a serial interface. File transfers between PC and module are greatly facilitated by this suite. The MES is installed to "Program Files\Cinterion\Module Exchange Suite".

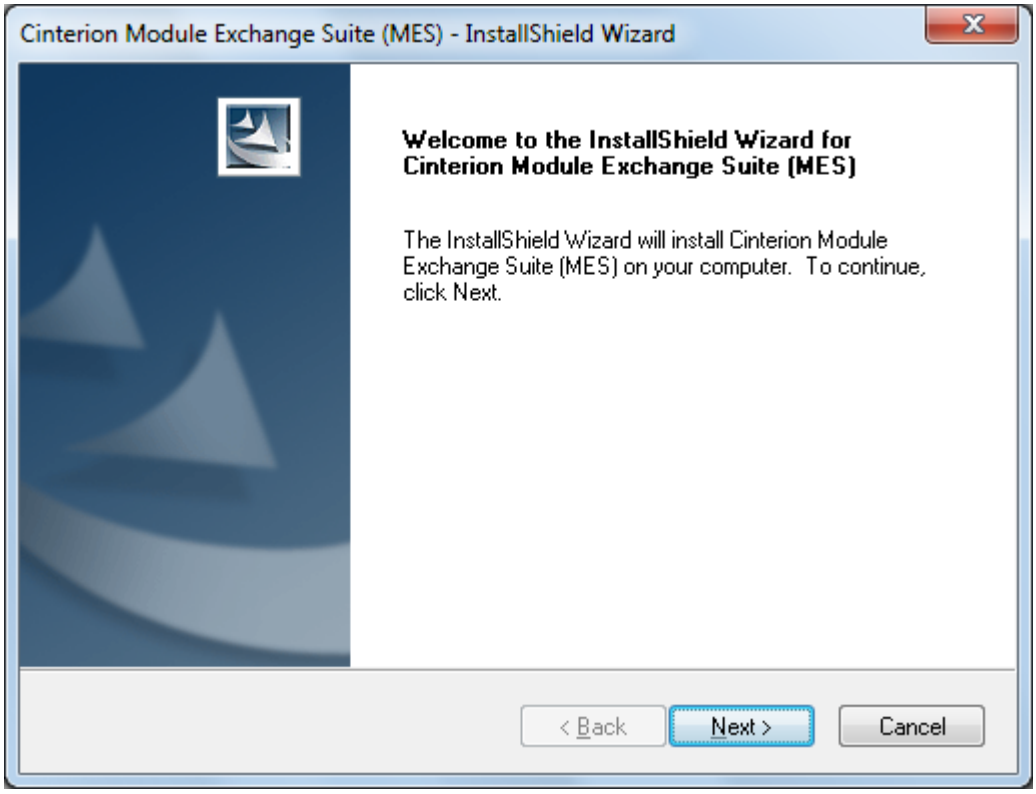


Figure 3: Module Exchange Suite - InstallShield Wizard

5. Read the MES license agreement. If you accept the agreement, press Yes to continue with the installation.
6. Click Finish to complete the MES installation.

7. Install the IMP debug connection as part of the CMTK installation.

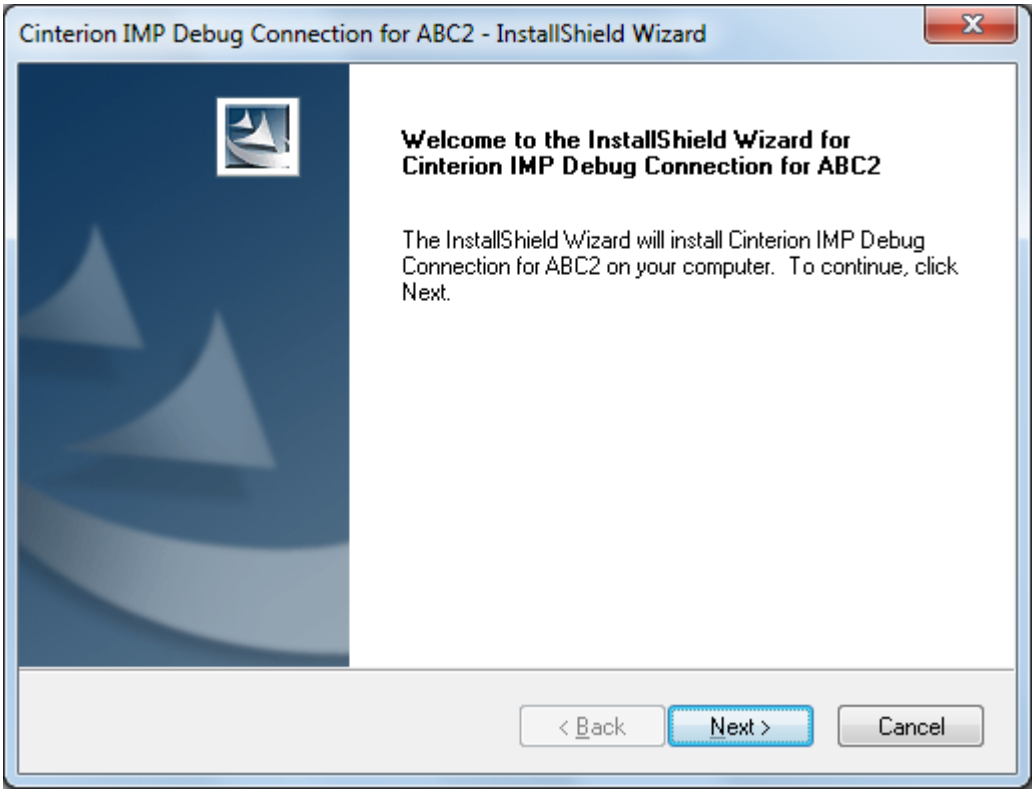


Figure 4: IMP Debug Connection - InstallShield Wizard

8. Continue with installation even if software did not pass the Windows logo test.
9. Scan COM ports for available Java module. The scan may be skipped and can be repeated later as part of a repair installation. This is done by selecting the Cinterion IMP Debug Connection from Control Panel --> Add or Remove Programs and clicking Change.

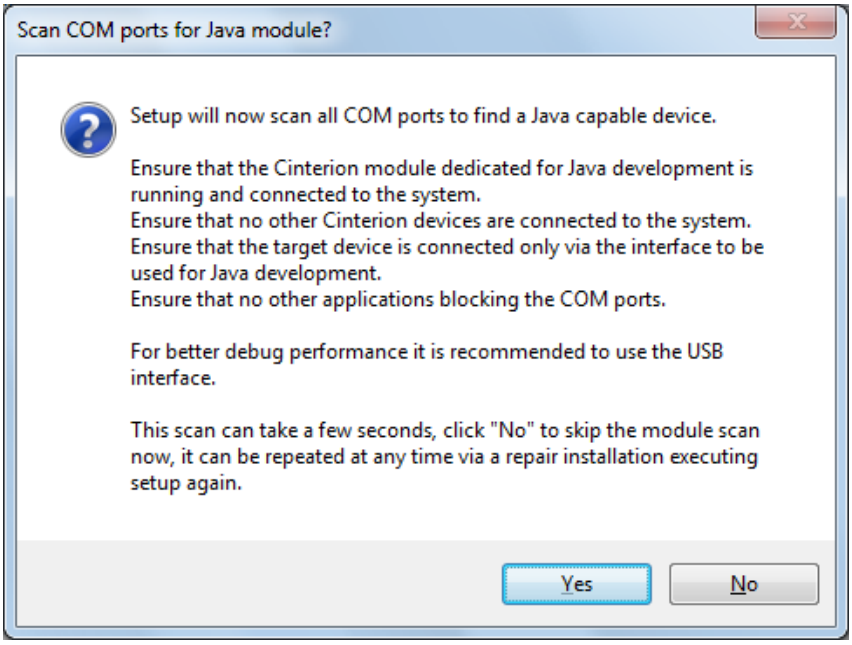


Figure 5: Scan COM ports for available JAVA module

10. Click Finish to complete the IMP Debug Connection installation.

11. Scan system for supported Java IDEs to automatically integrate the WTK into. Please ensure that none of the possibly installed Java IDEs is running before the scan is started. The scan may be skipped and can be repeated later as part of a repair installation. This is done by selecting the Cinterion ABC2 Software Development Kit from Control Panel --> Add or Remove Programs and clicking Change.

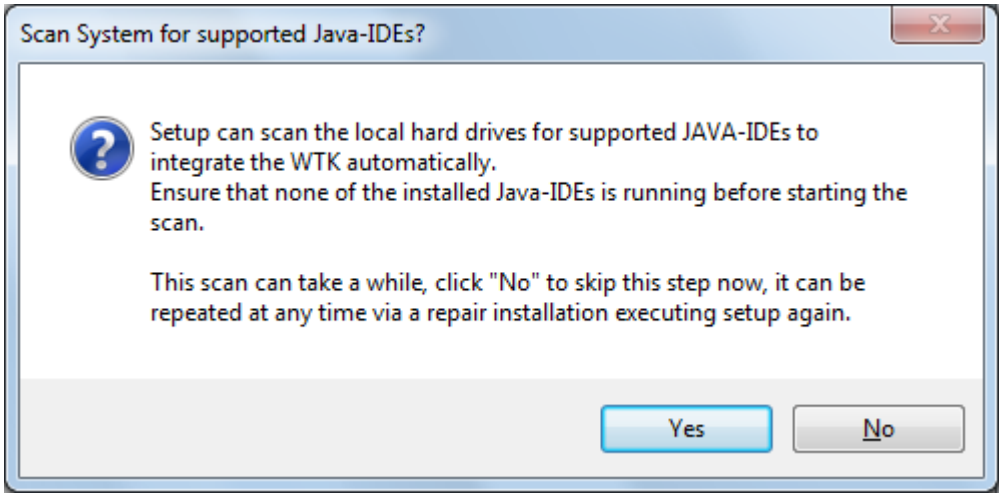


Figure 6: Scan for supported JAVA IDEs

12. If the above scan did not deliver any supported Java IDE, it is possible to install the Eclipse IDE ("Pulsar") provided on the installation CD.

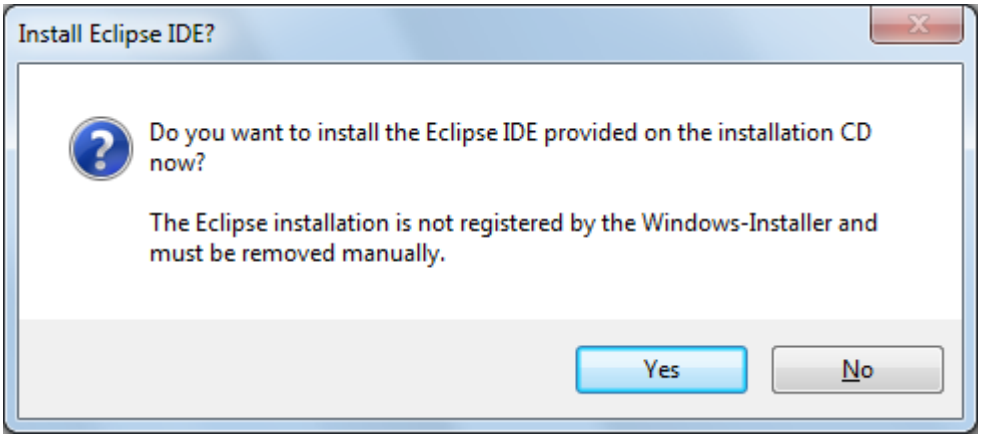


Figure 7: Query to install Eclipse IDE as part of CMTK installation

13. Click Finish to complete the CMTK installation.

3.4 CMTK Uninstall

The CMTK install package comes with an uninstall facility. The entire CMTK or parts of the package can be removed. To start the uninstall facility, open the Control Panel, select Add/Remove Programs, select the desired CMTK, e.g. Cinterion ABC2 Software Development Kit and follow the instructions. The standard modem and Dial-Up Network connection (DUN) are uninstalled automatically.

The Module Exchange Suite (MES) is not uninstalled automatically with the CMTK. To uninstall the MES as well, please run the MES uninstall facility. To run the uninstall program, open the Control Panel, select Add/Remove Programs, select Cinterion Module Exchange Suite (MES) and follow the instructions. The MES might still be used by other CMTK versions and should in this case not be uninstalled.

Please keep in mind, that standard modem (or USB modem) and Dial-Up Network connection are required for a proper working of the CMTK on-device debugging.

3.5 Upgrades

The CMTK can be modified, repaired or removed by running the setup program on the Installation CD.

4 Software Platform

In this chapter, we discuss the software architecture of the CMTK and the interfaces to it.

4.1 Software Architecture

The CMTK enables a customer to develop a Java application on a PC and have it be executable on the Java enabled module. The application is then loaded onto the module. The platform comprises:

- Java™ Micro Edition (Java ME™), which forms the base of the architecture.
  The Java ME™ is provided by SUN Microsystems, http://java.sun.com/javame/. It is specifically designed for embedded systems and has a small memory footprint. The ME uses:
  CLDC 1.1 HI, the connected limited device configuration hot spot implementation.
  IMP-NG, the information module profile 2nd generation, this is for the most part identical to MIDP 2.0 but without the lcdui package.
- Additional Java virtual machine interfaces:
  AT Command API
  File I/O API
  The data flow through these interfaces is shown in Figure 9 and Figure 24.
- Memory space for Java programs:
  Flash File System: around 1.7 MB (8 MB for TC65i-X and EGS5-X partitioned into 4 MB on A:\ drive and 4 MB on C:\ drive. The C:\ drive is intended as a temporary storage when updating the EGS5-X firmware over the air (FOTA)).
  RAM: around 400kB (1.7 MB for TC65i-X and EGS5-X)
  Application code and data share the space in the flash file system and in RAM.
- Additional accessible periphery for Java applications
  - A maximum of ten digital I/O pins usable, for example, as:
    Output: status LEDs
  - Input: Emergency Button
  - One I2C/SPI Interface.
  - One Digital Analog Converter and two Analog Digital Converters.
  - Serial interface (RS-232 API): This standard serial interface could be used, for example, with an external current meter.
    For detailed information see Section 4.2.

4.2 Interfaces

4.2.1 ASC0 - Serial Device

ASC0, an Asynchronous Serial Controller, is a 9-wire serial interface. It is described in [2]. Without a running Java application the module can be controlled by sending AT commands over ASC0. Furthermore, ASC0 is designed for transferring files from the development PC to the module. When a Java application is started, ASC0 can be used as an RS-232 port or/and System.out. Refer to [3] for details.

4.2.2 General Purpose I/O

There are ten I/O pins that can be configured for general purpose I/O. One pin can be configured as a pulse counter. All lines can be accessed under Java by AT commands or a Java API. See [1] and [2] for information about usage and startup behavior.

4.2.3 DAC/ADC

There are two analogue input lines and one analogue output line. They are accessed by AT commands or via a Java API. See [1] and [2] for details.

4.2.4 ASC1

ASC1 is the second serial interface on the module. This is a 4-pin interface (RX, TX, RTS, CTS). It can be used as a second AT interface when a Java application is not running or by a running Java application as RS-232 port or/and System.out.

4.2.5 Digital Audio Interface (DAI)

The ME has a seven-line serial interface with one input data clock line and input/output data and frame lines to support the DAI. Refer to [1] and [2] for more information.

4.2.6 I2C/SPI

There is a 4 line serial interface which can be used as I2C or SPI interface. It is described in [2]. The AT^SSPI AT command configures and drives this interface. For details see [1]. I2C and SPI are also accessible from a Java API.

4.2.7 JVM Interfaces

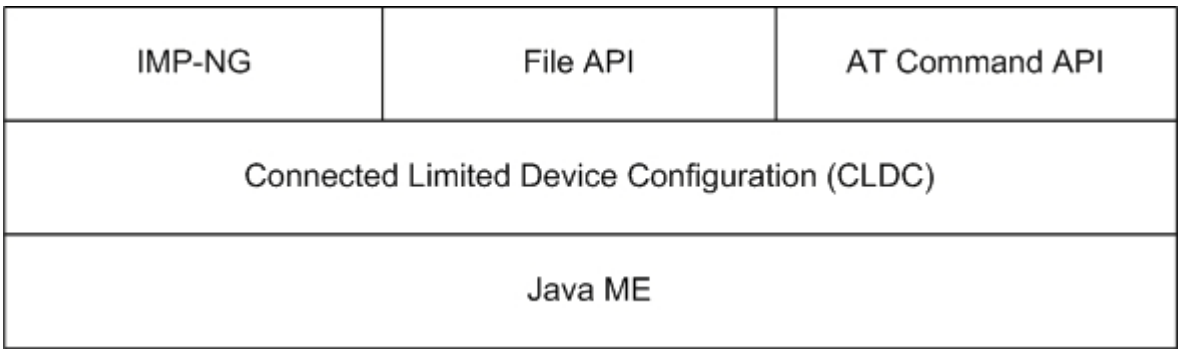


Figure 8: Interface Configuration

Java ME, CLDC and MIDP were implemented by SUN. IMP-NG is a stripped down version of MIDP 2.0 and does not include the graphical interface LCDUI. There are also additional APIs like the File I/O and the AT command API. Documentation for Java ME and CLDC can be found at http://java.sun.com/javame/. Documentation for the other APIs is found in [3].

4.2.7.1 IP Networking

IMP-NG provides access to TCP/IP similarly to MIDP 2.0.

Because the used network connection, CSD or GPRS, is fully transparent to the Java interface, the CSD and GPRS parameters must be defined separately either by the AT command AT^SJNET [1] or by parameters given to the connector open method, see [3].

4.2.7.2 Media

The playTone method and the tone sequence player are supported. For optimum performance use notes in the range of 48 to 105. Tones outside this range are affected by audio hardware filtering (see [2]).

4.2.7.3 Other Interfaces

Neither the PushRegistry interfaces and mechanisms nor any URL schemes for the Platform-Request method are supported. See [3].

4.3 Data Flow of a Java Application Running on the Module

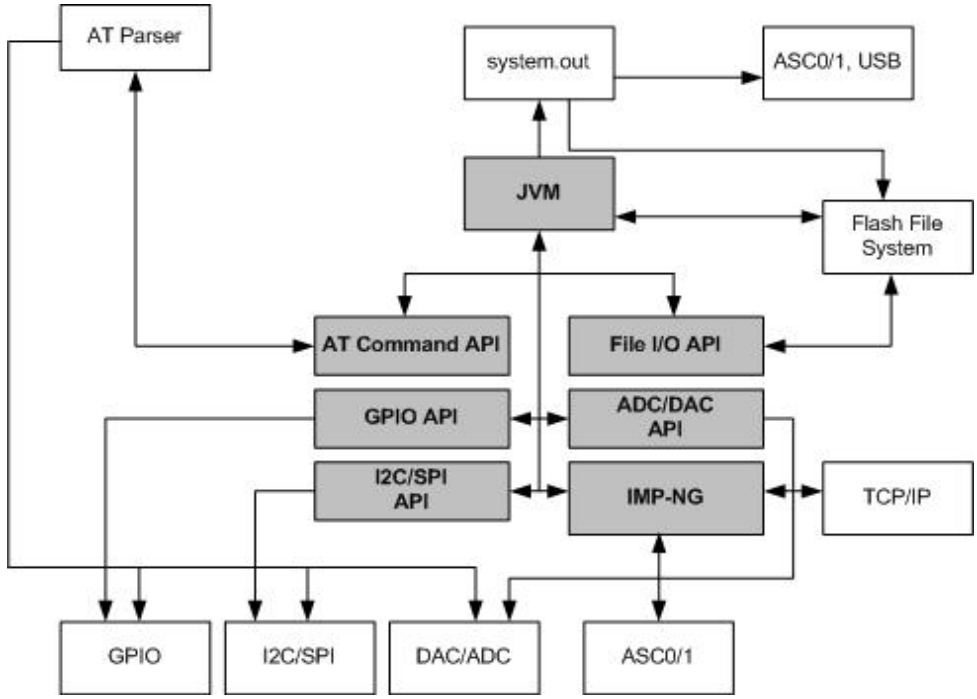


Figure 9: Data flow of a Java application running on the module.

The diagram shows the data flow of a Java application running on the module. The data flow of a Java application running in the debug environment can be found in Figure 24.
The compiled Java applications are stored as JAR files in the Flash File System of module. When the application is started, the JVM interprets the JAR file and calls the interfaces to the module environment.

The module environment consists of the:
- Flash File System: available memory for Java applications
- TCP/IP: module internal TCP/IP stack
- GPIO: general purpose I/O
- ASC0: Asynchronous serial interface 0
- ASC1: Asynchronous serial interface 1
- I2C: I2C Bus interface
- SPI: Serial Peripheral Interface
- DAC: digital analog converter
- ADC: analog digital converter
- AT parser: accessible AT parser

The Java environment on the module consists of the:
- JVM: Java Virtual Machine
- AT command API: Java API to AT parser
- File API: Java API to Flash File System
- IMP-NG: Java API to TCP/IP and ASC0/1
- GPIO API: Java API to GPIO pins and pulse counter
- I2C/SPI API: Java API to access I2C/SPI Bus
- ADC/DAC API: Java API to access ADC and DAC
- Watchdog API: Java API to HW watchdog
- Bearer Control API: Java API for bearer state information and hang-up.

4.4 Handling Interfaces and Data Service Resources

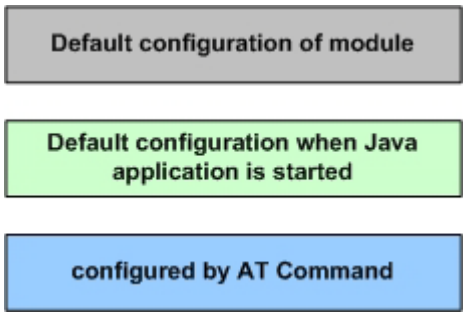
To develop Java applications the developer must know which resources, data services and hardware access are available.

- There are multiple AT parsers available
- There is hardware access over
  - two serial interfaces: ASC1 and ASC0 (both fully accessible).
  - general purpose I/O. To configure the hardware access, please refer to [1] and [2].
  - I2C/SPI
  - All restrictions of combinations are described in Section 4.4.1
- A Java application has:
  - instances of the AT command class, one with CSD and the others without, each of which would, in turn, be attached to one of the AT parsers.
  - two instances of access to a serial interface, ASC0 and ASC1, through the CommConnection API. Access to the control lines of these interfaces through CommConnectionControlLines.
  - System.out over any serial interface or into the file system

4.4.1 Module States

The module can exist in the following six states in relation to a Java application, the serial interfaces, GPIO and I2C/SPI. See [1] for information about the AT commands referenced. A state transition diagram is shown in Figure 13.

This section shows how Java applications must share AT parsers, GPIO pins and I2C/SPI resources. DAC, ADC and DAI are not discussed here.
Color legend for the following figures:



4.4.1.1 State 1: Default – No Java Running

This is the default state. The Java application is inactive and there is an AT interface with CSD on ASC0 as well as ASC1. The initial state of the pins is according to [4].

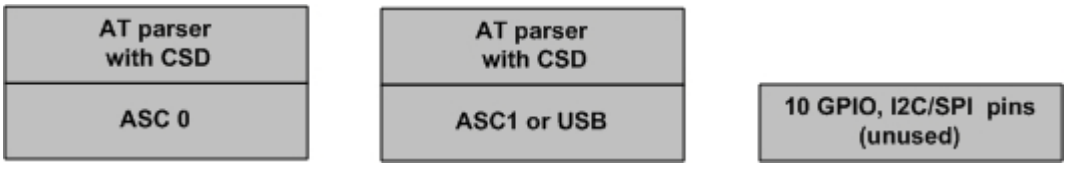


Figure 10: Module State 1

4.4.1.2 State 2: No Java Running, General Purpose I/O and I2C/SPI

The Java application is inactive. There is an AT parser with CSD on ASC0 as well as ASC1. Up to ten I/O pins are used as general purpose I/O plus a I2C/SPI interface. The pins are configured *by* AT^SCPIN and AT^SSPI (refer [1]).

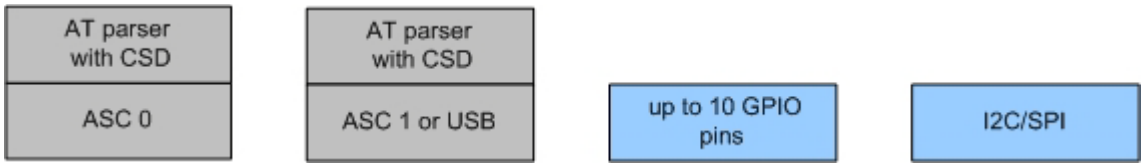


Figure 11: Module State 2

4.4.1.3 State 4: Default – Java Application Active

The Java application is active and ASC0 and ASC1 are used as System.out and/or CommConnection. Java instances of AT commands are connected to the available AT parsers. The Java application is activated with *AT^SJRA* (refer to [1]) or autostart.

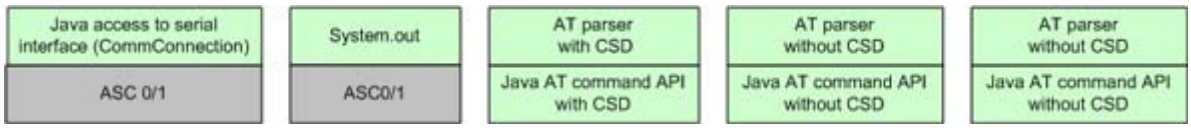


Figure 12: Module State 4

4.4.1.4 State 5: Java Application Active, General Purpose I/O and I2C/SPI

The Java application is active and ASC0 and ASC1 are used as System.out and/or CommConnection. The Java application is activated with *AT^SJRA.* The I/O pins are configured with AT^SCPIN, the I2C/SPI interface with AT^SSPI. Refer to [1] for AT command details.

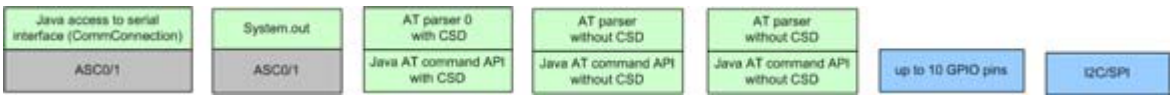


Figure 13: Module State 5

4.4.2 Module State Transitions

Figure 14: Module State Transition Diagram

Note: No AT parser is available over serial interface ASC0 or ASC1 while a Java application is running on the module.

5 Maintenance

The basic maintenance features of the ME are described below. Explicit details of these functions and modes can be found in [1] and [2].

5.1 IP Service

Apart from the standard Java IP networking interfaces (UDPDatagramConnection, SocketConnection, ...) the ME also supports a set of Internet Services controlled by AT command. There are some correlations between the Java and the AT IP Services.

- The connection profile 0 is also used by Java: when Java starts up a networking connection it tries to set and activate connection profile 0 with the parameters configured by AT^SJNET or in the connector.open method.
- Java tries to (re-)use an active Internet Service profile: if using connection profile 0 fails, because e.g. this (or another) connection profile is already used by the Internet Services, Java networking also uses this, already active, profile.
- Deactivation of the connection profile happens when all applications are finished: Java has its networking idle time. For the Internet Services an inactivity timeout referred to as parameter <inactTO> is available (configurable by AT^SICS and AT^SCFG).

So that means that Java networking and AT Internet Services can be used in parallel but care has to be taken about configuring and activation of the connection profile. It is recommended to use connection profile 0 for the Internet Services and set the parameters to the same values as the Java networking parameters. This way it makes no difference whether the connection is activated by the Internet Services or Java.
There are some aspects which have to be kept in mind for all IP Services (Java and AT command):

- When an open TCP connection is cut (e.g. the other side dies/is switched off) it takes around 10 minutes during which retransmissions are send, until the situation is detected as an error (in Java an exception is thrown).
- The number of IP services used in parallel should be kept small. An active IP service uses up resources and may deteriorate the overall performance.
- If a user rapidly closes and opens TCP/IP connections (e.g. SocketConnection, HTTPConnection), a ConnectionNotFoundException reading "No buffer space available" may be thrown, explaining that all TCP/IP socket resources are exhausted. In the worst case, opening further TCP/IP connections is locked for up to 60 seconds.
- If a service is re-opened on the same port shortly after having closed the ServerSocketConnection, the ServerSocketConnection may not work properly. To ensure that the service works correctly the host is required to wait at least two minutes before reopening a server.
- For information about the bearer state, use the specific IP service command AT^SICI and, in addition, the general network commands AT+COPS and/or AT+CREG.
- When trying to start the bearer when it is still in the process of shutting down, e.g. right after a "network idle timeout" an IOException is thrown. Either use a delay or wait for bearer state to actually say "down".

Some Java products feature the BearerControl class. This class provides bearer state information and a method to hang-up.

5.2 Remote SIM Access

While Remote SIM Access (RSA) is normally closely coupled with the GSM 07.10 multiplexer there are some things to keep in mind when using it with Java.

- Java must not be started when RSA and/or the GSM 07.10 multiplexer is activated.
- When activating the RSA mode (AT^SRSA) via a Java AT Command channel while Java is running the parameter <muxChan> of the AT^SRSA command is ignored and RSA is activated on the channel where the command was issued. The Input- and Outputstream can then be used to transfer RSA protocol data.

5.3 Power Saving

The module supports several power saving modes which can be configured by the AT command AT+CFUN [1]. Power saving affects the Java application in two ways. First, it limits access to the serial interface (RS-232-API) and the GPIO pins. Second, power saving efficiency is directly influenced by the way a Java application is programmed.

Java hardware access limitations:

- In NON-CYCLIC SLEEP mode (cfun=0) the serial interface cannot be accessed. Toggling RTS does end NON-CYCLIC SLEEP mode. In CYCLIC SLEEP mode (CFUN=7 or 9) the serial interface can be used with hardware flow control (CTS/RTS).
- In all SLEEP modes the GPIO polling frequency is reduced so that only signal changes which are less than 0.2Hz can be detected properly. Furthermore, the signal must be constant for at least 2.7s to detect any changes. For further details see AT^SCPOL in [1] or refer to [2].

Java power saving efficiency:

- As long as any Java thread is active, power consumption cannot be reduced, regardless whether any SLEEP mode has been activated or not. A Java application designed to be power efficient should not have any unnecessarily active threads (e.g. no busy loops). Threads waiting in a blocking method (e.g. read) do not hinder power saving.
- When using networking functionality close all connectors and hang-up the bearer manually (using ATH for circuit switched connections) every time you intend to reduce power consumption. Disable the network idle timeout (=0).

When going to low power mode there always might be a transition time of around 10s till low power consumption state is reached.

5.3.1 Power Saving While GPRS is Active

It is also possible that the module saves power while the GPRS PDP context is activated but without ongoing data traffic. To save power in GPRS mode it is required to enable a power saving mode (e.g. AT+CFUN=9) and put all threads into a sleep state. Also, data connections shall be in a blocking read (serial interface and/or GPRS connections).

5.4 Charging

Please refer to [1] and [2] for general information about charging. Charging can be monitored by the running Java application. The JVM is active in Charge mode and in Charge-Only mode if autostart is activated. Only a limited number of AT commands are available when the module is in Charge-Only mode. A Java application must be able to handle the Charge-Only mode and reset the module to reinstate the normal mode. See [2] for information about the Charge-Only mode. The Charge-Only mode is indicated by URC "^SYSSTART CHARGE-ONLY MODE".

Note: When a Java application is started in Charge-Only mode only AT Command APIs without CSD are available. The mode-indicating URC is created after issuing the very first AT command on any opened channel. To read the URC it is necessary to register a listener (see [3]) on this AT command API instance before passing the first AT command.

5.5 Airplane Mode

The main characteristic of this mode is that the RF is switched off and therefore only a limited set of AT commands is available. The mode can be entered or left using the appropriate AT^SCFG command. This AT command can also be used to configure the airplane mode as the standard startup mode, see [2]. The JVM is started when autostart is enabled. A Java application must be able to handle this mode. The airplane mode is indicated by URC "SYSSTART AIRPLANE MODE". Since the radio is off all classes related to networking connections, e.g. SocketConnection, UDPDatagramConnection, SocketServerConnection, HTTPConnection, will throw an exception when accessed.

5.6 Alarm

The ALARM can be set with the AT+CALA command. Please refer to the AT Command Set [1] and Hardware Interface Description [2] for more information. One can set an alarm, switch off the module with AT^SMSO, and have the module restart at the time set with AT+CALA. When the alarm triggers the module restarts in a limited functionality mode, the "airplane mode". Only a limited number of AT commands are available in this mode, although the JVM is started when autostart is enabled. A Java application must be able to handle this mode and reset the module to reinstate the normal mode. The mode of a module started by an alarm is indicated by the URC "^SYSSTART AIRPLANE MODE".

Note: For detailed information which functionality is available in this mode see [1] and [2]. The mode indicating URC is created after issuing the very first AT command on any opened channel.

5.7 Shutdown

If an unexpected shutdown occurs, data scheduled to be written will get lost due to a buffered write access to the flash file system. The best and safest approach to powering down the module is to issue the AT^SMSO command. This procedure lets the module log off from the network and allows the software to enter a secure state and save all data. Further details can be found in [2].

5.7.1 Automatic Shutdown

The ME is switched off automatically in different situations:
- under- or overtemperature
- under- or overvoltage

Appropriate warning messages transmitted by the ME to the host application are implemented as URCs. To activate the URCs for temperature conditions use the AT^SCTM command. Undervoltage and overvoltage URCs are generated automatically when fault conditions occur.

For further detail refer to the commands AT^SCTM and AT^SBC described in the AT Command Set [1]. In addition, a description of the shutdown procedures can be found in [2].

5.7.2 Manual Shutdown

The module can be switched off manually with the AT command, AT^SMSO. In this case the midlets destroyApp method is called and the application has 5s time to clean up and call the notifydestroy method. After the 5s the VM is shut down.

5.7.3 Restart after Switch Off

When the module is switched off without setting an alarm time (see the AT Command Set [1]), e.g. after a power failure, external hardware must restart the module with the Ignition line (IGT). The Hardware Interface Description [2] explains how to handle a switched off situation.

5.7.4 Watchdog

The Watchdog class allows to access the HW watchdog of the system from application level. Depending on the setting (at^scfg) the userware watchdog can do nothing, switch-off or reboot the system.

5.8 Special AT Command Set for Java Applications

For the full AT command set refer to [1]. There are differences in the behaviour AT commands issued from a Java application in comparison to AT commands issued over a serial interface.

5.8.1 Switching from Data Mode to Command Mode

Cancellation of the data flow with "+++" is not available in Java applications, see [1] for details. To break the data flow use *breakConnection().* For details refer to [3].

5.8.2 Mode Indication after MIDlet Startup

After starting a module without autobauding on, the startup state is indicated over the serial interface. Similarly, after MIDlet startup the module sends its startup state (^SYSSTART, ^SYSSTART AIRPLANE MODE etc.) to the MIDlet. This is done via a URC to the AT Command API instance which executes the very first AT Command from within Java. To read this URC it is necessary to register a listener (see [3]) on this AT Command API instance before passing the first AT Command.

5.8.3 Long Responses

The AT Command API can handle responses of AT commands up to a length of 1024 bytes. Some AT commands have responses longer than 1024 bytes, for these responses the Java application will receive an Exception.

Existing workarounds:
- Instead of listing the whole phone book, read the entries one by one
- Instead of listing the entire short message memory, again list message after message
- Similarly, read the provider list piecewise
- Periods of monitoring commands have to be handled by Java, e.g. AT^MONI, AT^SMONG. These AT commands have to be used without parameters, e.g. for AT^MONI the periods must be implemented in Java.

5.8.4 Configuration of Serial Interface

While a Java application is running on the module, only the AT Command API is able to handle AT commands. All AT commands referring to a serial interface are ignored. This includes the commands:
- AT+IPR (sets a fixed local bit rate)
- AT\Q1, AT\Q2 and AT\Q3 (sets type of flow control)

If Java is running, the firmware will ignore any settings from these commands. Responses to the read, write or test commands will be invalid or deliver „ERROR".

Note: When a Java application is running, all settings of the serial interface are done with the class CommConnection. This is fully independent of any AT commands relating to a serial interface. However, the following restrictions apply in configuring the serial interface: Baudrate: Only 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 115200, 230400 and 460800 are supported; Stopbits: Only "1" is supported; Parity: "7N1" is not supported.

5.8.5 Java Commands

There is a small set of special Java AT commands:
- AT^SJRA, start a Java application
- AT^SJNET, configuration of Java networking connections
- AT^SJOTAP, start and configuration of over the air provisioning
- AT^SJSEC, security configuration

Refer to the AT command Set [1].

5.8.6 AutoExec Function

Under Java, the AutoExec function's AT command execution has some restrictions (see [1] for details on the AutoExec function). If an AT command is executed, neither URC nor command response are indicated to the Java application. Therefore, it is recommended to use the AutoExec function under Java only with commands that shut down the module, e.g., with AT+CFUN=0,1, as a watch dog. Any other AT command execution should be implemented with Java means and the ATCommand class.

5.9 System Out

Any output printed to the System.out stream by a Java application can be redirected to one of the serial interfaces, a file, a "NULL" device (i.e. the output will be discarded) or a UDP socket for using the debugger from an IDE. The configuration can be done at any time using the AT command AT^SCFG (see [1] for details) and is non-volatile.

5.9.1 Serial interfaces

System.out can be written to any of the serial interfaces ASC0, ASC1 or USB. If System.out is redirected to one of the interfaces used for the Java CommConnection, the interface will be shared between System.out and the CommConnection. This will result in mixed output, if data is written to the CommConnections OutputStream and something is printed via System.out at the same time.

Using System.out and CommConnection on the same serial interface may be done if a device connected to the serial port is only transmitting data to the module. It is recommended to ensure already in the HW design that output from the module cannot be transferred to a connected device.

5.9.2 File

The System.out print can be redirected into log files within the module's flash file system. The output will be written alternatingly into two files which can be concatenated afterwards to have a single log file.

Writing the output to a file will slow down the virtual machine. To reduce the impact of logging the output may be written first to a buffer before it is written to the file (buffered mode). The buffered output is written either if the buffer is filled or after 200 ms. If the buffer is not used (secure mode) the output is written directly to the file. Because excessive writing to the module's flash file system decreases the life time of the flash memory, we recommend using the System.out to file redirection only during development phases.

5.9.3 UDP

Redirection to a UDP socket is used in conjunction with the debugger. UDP is used by default when using on-device-debugging (Chapter 10). This can be changed by editing the emulator's ini file or by using the appropriate command line parameter (Section 10.4).

If the output is redirected to an UDP socket, any changes of the System.out configuration are ignored while the Virtual Machine is running. The UDP Socket settings will not be stored in the module.

5.10 GPIO

The GPIO Java API (classes: InPort, OutPort, InPortListener, StartStopPulseCounter, LimitPulseCounter and LimitPulseCounterListener) is a replacement for the GPIO AT commands (AT^SPIO, AT^SCPIN, ...). Using these classes frees up AT command resources and improves performance. Both APIs can only be used alternatively. Of course the basic system characteristics of a poll interval of 4.6ms (in SLEEP mode up to 2.7s) for pin state change detection does also apply here.

5.11 Restrictions

5.11.1 Flash File System

The maximum length of a complete path name, including the path and the filename, is limited by the Flash file system on the module to 124 characters. It is recommended that names of classes and files be distinguished by more than case sensitivity.
Avoid using any blank in the names of JAR or JAD files. Otherwise the file explorer might not recognize the names. If the OTAP server adds a blank into the filenames, problems with OTAP will occur.

5.11.2 Memory

The CLDC 1.1 HI features a just-in-time compiler. This means that parts of the Java byte code which are frequently executed are translated into machine code to improve efficiency. This feature uses up RAM space. There is always a trade off between code translation to speed up execution and RAM space available for the application.

5.11.3 JAD File Size

The Java Application Descriptor (JAD) File can be used to pass configuration data to the midlet. JAD file sizes of up to 30 KByte are allowed.

5.12 Performance

The performance study was focused on comparable performance values under various circumstances.

5.12.1 Java

This section gives information about the Java command execution throughput ("jPS"= Java statements per second). The scope of this measurement is only the statement execution time, not the execution delay (Java command on AT interface → Java instruction execution → reaction on GPIO).

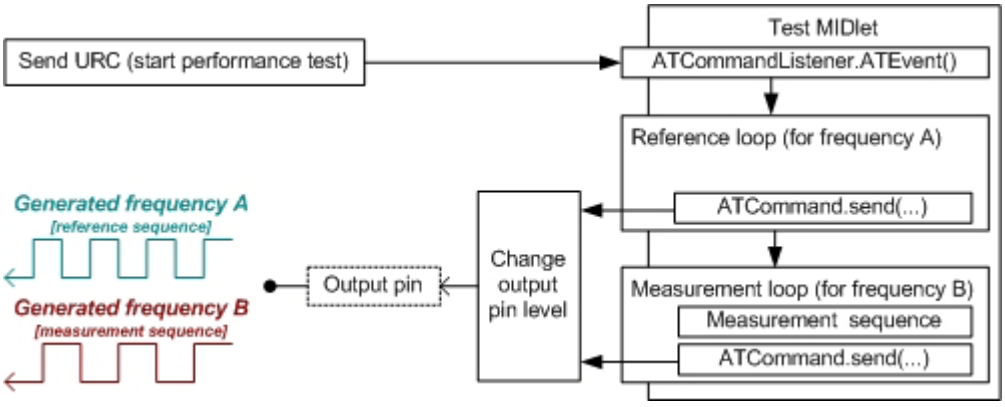


Figure 15: Test case for measuring Java command execution throughput

The following Java instruction was used for calculation of the typical jPS:
value = (2 x number of calculation statements) /
((1 / frequencyB) - (1 / frequencyA));

Measurement and calculation were done using:
- **duration of each loop** = 600 s
- **number of calculation statements** = 50 "result=(CONSTANT_VALUE/variable_value);"-Instructions (executed twice per pin cycle)
- **frequencyA** as measured with a universal counter
- **frequencyB** as measured with a universal counter

The reference loop has the same structure as the measurement loop except that the measurement sequence is moved.

State	jPS-Value (mean)
ME in IDLE mode / Not connected	~49000
CSD connection	~46000

Since only a small amount of Java code is executed in this test, it is easily optimized by the CLDC's HI just-in-time compiler. More complex applications might not reach that execution speed.

5.12.2 Pin I/O

The pin I/O test was designed to find out how fast a Java MIDlet can process URCs caused by Pin I/O and react to these URCs. The URCs are generated by feeding an input pin with an external frequency. As soon as the Java MIDlet is informed about the URC, it tries to regenerate the feeding frequency by toggling another output pin.

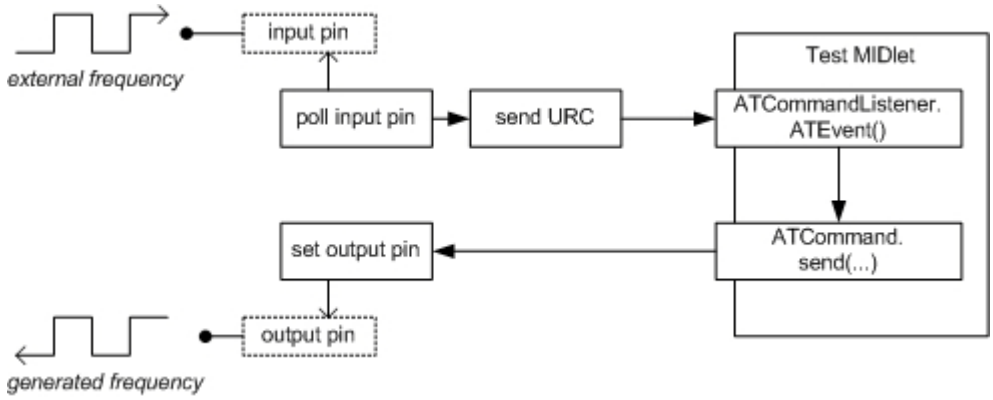


Figure 16: Test case for measuring Java MIDlet performance and handling pin-IO

The results of this test show that the delay from changing the state on the input pin to a state change on the output pin is at least around 60 ms, but that time strongly depends on the amount of garbage to collect and number of threads to be served by the virtual machine.
When doing this scenario using the Java GPIO API instead of ATCommand the delay goes down to around half this value.
But still pin I/O is not really suitable for generating or detecting frequencies.

5.12.3 Data Rates on RS-232 API

For details about the software platform and interfaces refer to Chapter 4, "Software Platform". This section summarises limitations and preconditions for performance when using the interface CommConnection from package com.cinterion.io[1](refer to [3]).

The data rate on RS232 depends on the size of the buffer used for reading from and writing to the serial interface. It is recommended that method *read (byte[ ] b)* be used for reading from the serial interface. The recommended buffer size is 2kbyte. To achieve error free data transmission the flow control on CommConnection must be switched on: <autorts> and <autocts>, the same for the connected device.

Different use cases are listed to give an idea of the attainable data rates. All applications for measurement use only one thread and no additional activities other than those described were carried out in parallel.

1. Up to TC65i v01.100 this package is named: com.siemens.icm.io.

5.12.3.1 Plain Serial Interface

Scenario: A device is connected to ASC0 (refer to Section 4.2.4). The Java application must handle data input and output streams. A simple Java application (with only one thread) which loops incoming data directly to output, reaches data rates up to 180kbit/s. Test conditions: hardware flow control enabled (<autorts> and <autocts>), 8N1, and baud rate on ASC0 set to 230kbaud (→ theoretical maximum: 184kbit/s net data rate).

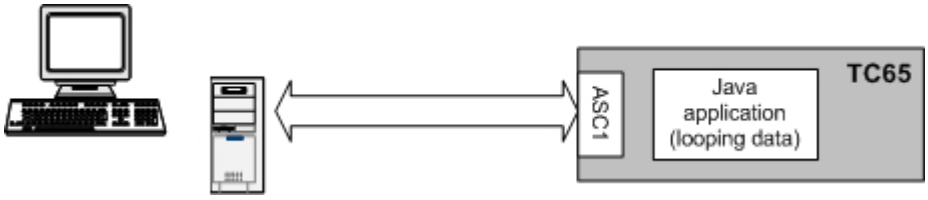


Figure 17: Scenario for testing data rates on ASC1

5.12.3.2 Voice Call in Parallel

Same scenario as in Section 5.12.3.1, but with a voice call added. The application reflects incoming data directly to output and, additionally, handles an incoming voice call. The data rates are also up to 180kbit/s. Test conditions: same as in Section 5.12.3.1.

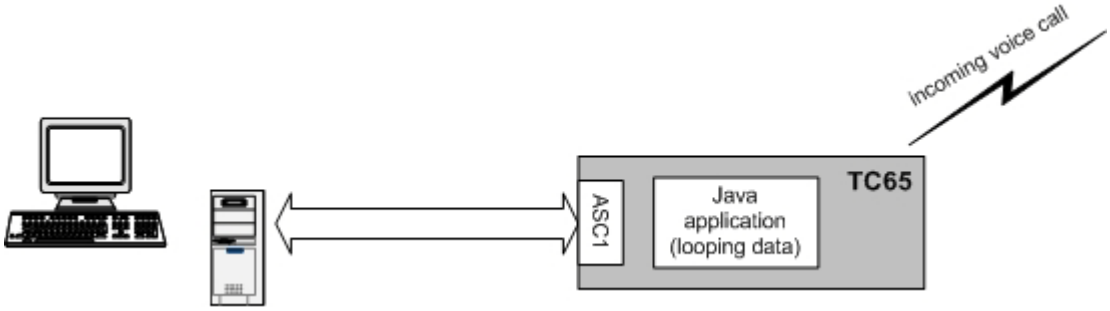


Figure 18: Scenario for testing data rates on ASC1 with a voice call in parallel

5.12.3.3 Scenarios with GPRS/EGDE Connection

The biggest challenges to the module performance are setting up a GPRS/EDGE connection, receiving data on javax.microedition.io interfaces and sending or receiving the data on the RS232 API with the help of a Java application.

5.12.3.4 Upload

The ME supports up to four uplink time slots for GPRS and up to two for EGDE. The Java application receives data over RS232 API and sends them over GPRS to a server.

Table 1: GPRS upload data rate with different number of timeslots, CS2

Upload data rate with x timeslots Coding scheme 2 [kbit/s]											
1 time-slot	**theor. Value[1]**	**% from theor. Value**	**2 time-slots**	**theor. Value[1]**	**% from theor. Value**	**3 time-slots**	**theor. Value[1]**	**% from theor. Value**	**4 time-slots**	**theor. Value[1]**	**% from theor. Value**
9	12	75%	15	24	63%	20	36	55%	16	48	33%

1. net transmission rates for LLC layer

Table 2: GPRS upload data rate with different number of timeslots, CS4

Upload data rate with x timeslots Coding scheme 4 [kbit/s]											
1 time-slot	**theor. Value[1]**	**% from theor. Value**	**2 time-slots**	**theor. Value[1]**	**% from theor. value**	**3 time-slots**	**theor. Value[1]**	**% from theor. value**	**4 time-slots**	**theor. Value[1]**	**% from theor. value**
13	20	65%	22	40	55%	20	60	33%	13	80	16%

1. net transmission rates for LLC layer

Table 3: EDGE upload data rate with two timeslots, CS5

Upload data rate with x timeslots Coding scheme 5 [kbit/s]		
2 timeslots	**theor. Value[1]**	**% from theor. Value**
33	44	73%

1. net transmission rates for LLC layer

Table 4: EDGE upload data rate with two timeslots, CS9

Upload data rate with x timeslots Coding scheme 9 [kbit/s]		
2 time-slots	**theor. Value[1]**	**% from theor. Value**
72	118	61%

1. net transmission rates for LLC layer

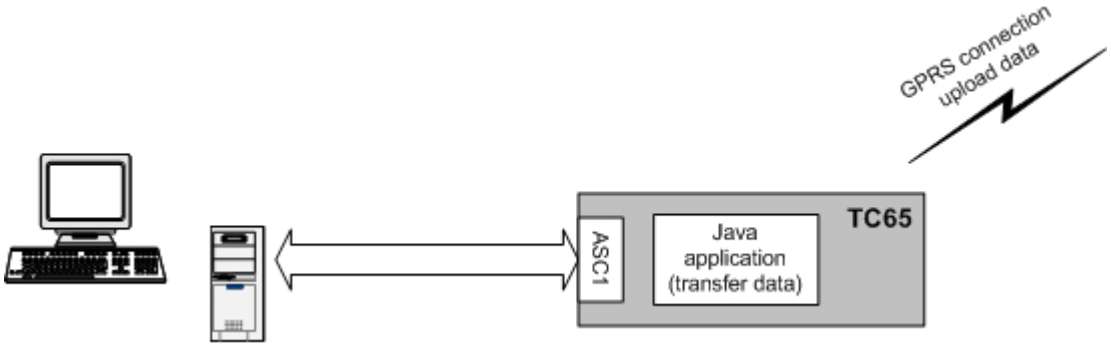


Figure 19: Scenario for testing data rates on ASC1 with GPRS data upload

5.12.3.5 Download

The data rate for downloading data over GPRS/EDGE depends on the number of assigned timeslots and the coding schemes given by the net. The ME supports up to four downlink time slots. For the measurements, the Java application receives data from the server over GPRS and sends them over RS232 to an external device.

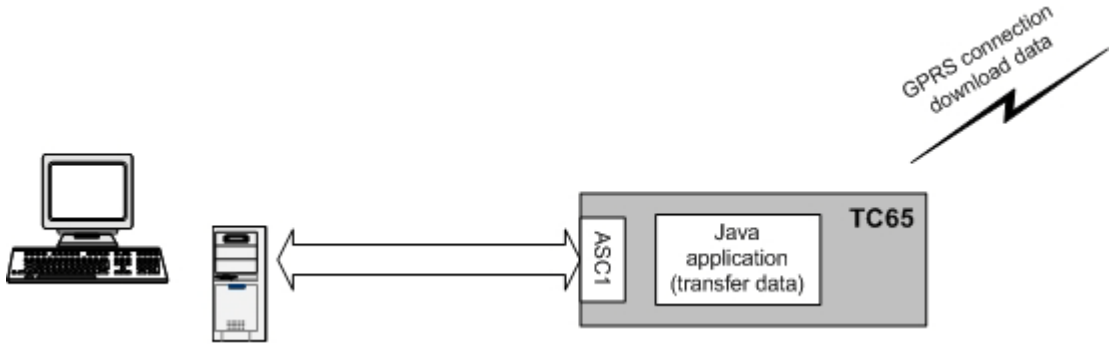


Figure 20: Scenario for testing data rates on ASC1 with GPRS data download

The tables below show the download data rates that can be achieved if hardware flow control is enabled on the CommConnection interface and the port speed is set to 460800.

Table 5: GPRS Download data rate with different number of timeslots, CS2

Download data rate with x timeslots Coding scheme 2 [kbit/s]											
1 time-slot	**theor. Value [1]**	**% from theor. Value**	**2 time-slots**	**theor. Value[1]**	**% from theor. Value**	**3 time-slots**	**theor. Value[1]**	**% from theor. Value**	**4 time-slots**	**theor. Value[1]**	**% from theor. Value**
11	12	91%	21	24	87%	29	36	81%	35	48	73%

1. net transmission rates for LLC layer

Table 6: GPRS Download data rate with different number of timeslots, CS4

Download data rate with x timeslots Coding scheme 4 [kbit/s]											
1 time-slot	**theor. Value [1]**	**% from theor. Value**	**2 time-slots**	**theor. Value[1]**	**% from theor. value**	**3 time-slots**	**theor. Value[1]**	**% from theor. value**	**4 time-slots**	**theor. Value[1]**	**% from theor. value**
17	20	85%	31	40	78%	35	60	58%	38	80	48%

1. net transmission rates for LLC layer

Table 7: EDGE Download data rate with different number of timeslots, CS5

Download data rate with x timeslots Coding scheme 5 [kbit/s]					
1 time-slot	**theor. Value [1]**	**% from theor. Value**	**4 time-slots**	**theor. Value[1]**	**% from theor. Value**
20	22	91%	78	89	87%

1. net transmission rates for LLC layer

Table 8: EDGE Download data rate with different number of timeslots, CS9

Download data rate with x timeslots Coding scheme 9 [kbit/s]					
1 time-slot	**theor. Value [1]**	**% from theor. Value**	**4 time-slots**	**theor. Value[1]**	**% from theor. value**
50	59	85%	184	236	77%

1. net transmission rates for LLC layer

5.13 System Time

When Java starts up, it initializes its time base from the system's real time clock. If the RTC is changed by AT command (AT+CCLK) later on, the Java time does not adjust. So, the time you get with (AT+CCLK) and the time you get with System.currentTimeMilis() may not necessarily be identical.

6 MIDlets

The Java ME™ Mobile Information Device Profile (MIDP) provides a targeted Java API for writing wireless applications. The MIDP runs on top of the Connected Limited Device Configuration (CLDC), which in turn, runs on top of the Java ME™. MIDP applications are referred to as MIDlets. MIDlets are controlled by the mobile device implementation that supports the CLDC and MIDP. Since IMP-NG is a subset of MIDP 2.0, IMP includes MIDlets. The MIDlet code structure is very similar to applet code. There is no main method and MIDlets always extend from the MIDlet class. The MIDlet class in the MIDlet package provides methods to manage a MIDlet's life cycle.

6.1 MIDlet Documentation

MIDP and MIDlet documentation can be found at http://wireless.java.sun.com/midp/ and in the html document directory of the wtk,
…\Cinterion\CMTK\<productname>\wtk\doc\index.html

6.2 MIDlet Life Cycle

The MIDlet life cycle defines the protocol between a MIDlet and its environment through a simple well-defined state machine, a concise definition of the MIDlet's states and APIs to signal changes between the states. A MIDlet has three valid states:

- *Paused* – The MIDlet is initialised and is quiescent. It should not be holding or using any shared resources.
- *Active* – The MIDlet is functioning normally.
- *Destroyed* – The MIDlet has released all of its resources and terminated. This state is only entered once.

State changes are controlled by the MIDlet interface, which supports the following methods:

- *pauseApp()* – the MIDlet should release any temporary resources and become passive.
- *startApp()* – the MIDlet starts its execution, needed resources can be acquired here or in the MIDlet constructor.
  Note: Take care that the startApp() method is always properly terminated before calling the destroyApp() method. For example, avoid that threads launched by startApp() enter a closed loop, and be sure that all code was entirely executed. This is especially important for OTAP, which needs to call destroyApp().
- *destroyApp()* **–** the MIDlet should save any state and release all resources
  Note: To destroy only the Java application without switching off the module, the destroyApp() method can be called explicitly. To destroy the Java application and switch off the module at the same time, it is sufficient to send the AT^SMSO command from somewhere in your code, because this procedure implies calling the destroyApp() method. Likewise, resetting the module with AT+CFUN=x,1 also implies calling the destroyApp() method. Note that AT+CFUN=x,1 will restart the module – to restart Java afterwards either use the autostart mode configured with AT^SCFG or restart Java with AT^SJRA.

  From this you can see that the commands AT^SMSO and AT+CFUN=x,1 should never be sent within the destroyApp() method. It is good practice to always call the notifyDestroyed() method at the end of your destroyApp method. And use the destroyApp method as single exit point of your midlet.

- *notifyDestroyed()* – the MIDlet notifies the application management software that it has cleaned up and is done.
  Note: the only way to terminate a MIDlet is to call notifyDestroyed(), but destroyApp() is not automatically called by notifyDestroyed(). You must not terminate your midlet (i.e. having no threads left) and not calling notifyDestroyed() before.
- *notifyPaused()* – the MIDlet notifies the application management software that it has paused
- *resumeRequest()* – the MIDlet asks application management software to be started again.

Table 9: A typical sequence of MIDlet execution

Application Management Software	MIDlet
The application management software creates a new instance of a MIDlet.	The default (no argument) constructor for the MIDlet is called; it is in the Paused state.
The application management software has decided that it is an appropriate time for the MIDlet to run, so it calls the MIDlet.startApp method for it to enter the Active state.	The MIDlet acquires any resources it needs and begins to perform its service.
The application management software no longer needs the application be active, so it signals it to stop performing its service by calling the *MIDlet.pauseApp* method.	The MIDlet stops performing its service and might choose to release some resources it currently holds.
The application management software has determined that the MIDlet is no longer needed, or perhaps needs to make room for a higher priority application in memory, so it signals the MIDlet that it is a candidate to be destroyed by calling the *MIDlet.destroyApp* method.	If it has been designed to do so, the MIDlet saves state or user preferences and performs clean up.

6.3 Hello World MIDlet

Here is a sample HelloWorld program.

```
/**
 * HelloWorld.java
 */

package example.helloworld;
import javax.microedition.midlet.*;
import java.io.*;

public class HelloWorld extends MIDlet {

  /**
   * HelloWorld - default constructor
   */
  public HelloWorld() {
    System.out.println("HelloWorld: Constructor");
  }

  /**
   * startApp()
   */
  public void startApp() throws MIDletStateChangeException {
    System.out.println("HelloWorld: startApp");
    System.out.println("\nHello World!\n");
    destroyApp();
  }

  /**
   * pauseApp()
   */
  public void pauseApp() {
    System.out.println("HelloWorld: pauseApp()");
  }


  /**
   * destroyApp()
   */
  public void destroyApp(boolean cond) {
    System.out.println("HelloWorld: destroyApp(" + cond + ")");
    notifyDestroyed();
  }
}
```

7 File Transfer to Module

7.1 Module Exchange Suite

The Module Exchange Suite allows you to view the Flash file system on the module as a directory from Windows Explorer. Make sure that the module is turned on and that one of the module's serial interfaces (ASC0, ASC1 or USB) is connected to the COM port that the Module Exchange Suite is configured to. The configured COM port can be checked or changed under Properties of the Module directory. Please note that the Module Exchange Suite can be used only if the module is in normal mode and the baud rate is configured to a fixed value of 921600, 460800, 230400, 115200, 57600, 38400 or 19200. Please also note that the use of the Module Exchange Suite resets the module's AT command settings to their factory defaults (just as the command AT&F would). Possible user defined profiles stored non-volatile with AT&W will have to be restored with ATZ after usage.

While running the module with the Module Exchange Suite, subdirectories and files can be added to the flash file system of module. Keep in mind that a maximum of 200 flash objects (files and subdirectories) per directory in the flash file system of the module is recommended.

7.1.1 Windows Based

The directory is called "Module" and can be found at the top level of workspace "MyComputer". To transfer a file to the module, simply copy the file from the source directory to the target directory in the "Module → Module Disk (A:)".

7.1.2 Command Line Based

A suite of command line tools is available for accessing the module's Flash file system. They are installed under Module Exchange Suite installation directory so that the tools are available from any directory. The module's file system is accessed with *mod:*. The tools included in this suite are MESdel, MEScopy, MESxcopy, MESdir, MESmkdir, MESrmdir, MESport, MESclose and MESformat. Entering one of these commands without arguments will describe the command's usage. The tools mimic the standard directory and file commands. A path inside the module's file system is identified by using "mod:" followed by the module disk which is always "A:" (e.g. "MESdir mod:a:" lists the contents of the module's root directory).

7.2 Over the Air Provisioning

See Chapter 8 for OTA provisioning.

7.3 Security Issues

The developer should be aware of the following security issues. Security aspects in general are discussed in Chapter 11.

7.3.1 Module Exchange Suite

The serial interface should be mechanically protected.

The copy protection rules for Java applications prevent opening, reading, copying, moving or renaming of JAR files. It is not recommended that the name of a Java application (for example <name>.jar) be used for a directory, since the copy protection will refuse access to open, copy or rename such directories.

7.3.2 OTAP

- A password should be used to update with OTA (SMS Authentication)
- Parameters should be set to fixed values (AT^SJOTAP) whenever possible so that they cannot be changed over the air.
- The HTTP server should be secure (e.g. access control via basic authentication).
- Ensure that the OTAP server does not add a blank to the names of JAR and JAD files, because this will cause problems with OTAP.

8 Over The Air Provisioning (OTAP)

8.1 Introduction to OTAP

OTA (Over The Air) Provisioning of Java Applications is a common practice in the Java world. OTAP describes mechanisms to install, update and delete Java applications over the air. The ME implements the Over The Air Application Provisioning as specified in the IMP-NG standard (JSR228).

The OTAP mechanism described in this document does not require any physical user interaction with the device; it can be fully controlled over the air interface. Therefore it is suitable for Java devices that are designed not to require any manual interaction such as vending machines or electricity meters.

8.2 OTAP Overview

To use OTAP, the developer needs, apart from the device fitted with the Java enabled module, an http server, which is accessible over a TCP/IP connection either over GPRS or CSD, and an SMS sender, which can send Class1, PID $7d short messages. This is the PID reserved for a module's data download.

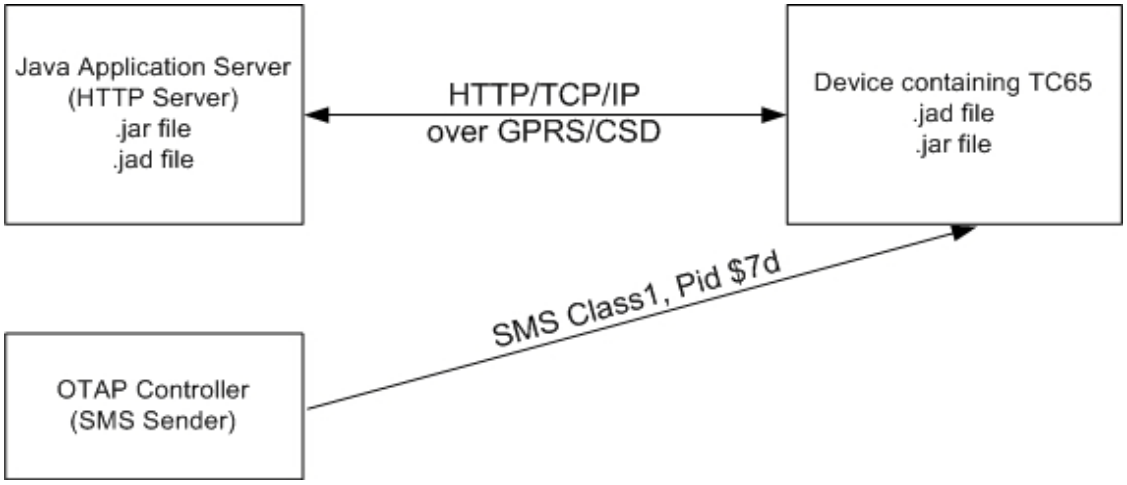


Figure 21: OTAP Overview

The Java Application Server (http Server) contains the .jar and the .jad file to be loaded on the device. Access to these files can be protected by http basic authentication.

The OTAP Controller (SMS Sender) controls the OTAP operations. It sends SMs, with or without additional parameters, to the devices that are to be operated. These devices then try to contact the http server and download new application data from it. The OTAP Controller will not get any response about the result of the operation. Optionally the server might get a result response over http.

There are two types of OTAP operations:
- Install/Update: A new JAR and JAD file are downloaded and installed.
- Delete: A complete application (.jar, .jad, all application data, its directory and up to 5 sub directory levels with all their files) is deleted.

8.3 OTAP Parameters

There is a set of parameters that control the OTAP procedures. These parameters can either be set by AT command (AT^SJOTAP, refer to [1] during the production of the device, or by SM (see Section 8.4) during operation of the device in the field. None of the parameters, which are set by AT command, can be overwritten by SM.

- JAD File URL: the location of the JAD file is used to install or update procedures. The JAD file needs to be located on the net (e.g. http://someserver.net/somefile.jad or http://192.168.1.2/somefile.jad).
- Application Directory: this is the directory where a new application (JAD and JAR file) is installed. The delete operation deletes this directory completely. When entering the application directory with AT^SJOTAP or a short message ensure that the path name is not terminated with a slash. For example, type "a:" or "a:/otap" rather than "a:/" or "a:/otap/". See examples provided in Chapter 6.
- http User: a username used for authentication with the http server.
- http Password: a password used for authentication with the http server.
- Bearer: the network bearer used to open the HTTP/TCP/IP connection, either GPRS or CSD.
- APN or Number: depending on the selected network bearer this is either an access point name for GPRS or a telephone number for CSD.
- Net User: a username used for authentication with the network.
- Net Password: a password used for authentication with the network.
- DNS: a Domain Name Server's IP address used to query hostnames.
- NotifyURL: the URL to which results are posted. This parameter is only used when the MIDlet-Install-Notify attribute or MIDlet-Delete-Notify attribute is not present in descriptor.

There are parameters that can only be set by AT command:

- SM Password: it is used to authenticate incoming OTAP SMs. Setting this password gives an extra level of security.
  Note: If a password set by AT command, all SMs must include this password.
- Ignore SM PID: when setting this the PID in received SMs is ignored.
- Hide HTTP authentication parameters: this allows to hide the http authentication parameters in the at^sjotap read command and the otap tracer.

Table 10: Parameters and keywords

Parameters	Max. Length AT	Keyword SM	Install/update	delete
JAD File URL	100	JADURL	mandatory	unused
Application Directory	50	APPDIR	mandatory	mandatory
HTTP User	32	HTTPUSER	optional	unused
HTTP Password	32	HTTPPWD	optional	unused
Bearer	--	BEARER	mandatory	optional/P
APN or Number	65	APNORNUM	mandatory for CSD	optional/P
Net User	32	NETUSER	optional	optional/P
Net Password	32	NETPWD	optional	optional/P
DNS	--	DNS	optional	optional/P
Notify URL	100	NOTIFYURL	optional	optional/P
SM Password	32	PWD	optional	optional

Table 10: Parameters and keywords

Parameters	Max. Length AT	Keyword SM	Install/update	delete
Ignore SM PID	--	--	--	--
Hide HTTP authentication parameters	--	--	--	--

The length of the string parameters in the AT command is limited (see Table 10), the length in the SM is only limited by the maximum SM length.

The minimum set of required parameters depends on the intended operation (see Table 10). "optional/P" indicates that this parameter is only necessary when a POST result is desired.

8.4 Short Message Format

An OTAP control SM must use a Submit PDU with Class1, PID $7d and 8 bit encoding. As a fallback for unusual network infrastructures the SM can also be of Class0 and/or PID $00. The content of the SM consists of a set of keywords and parameter values all encoded in ASCII format. These parameters can be distributed over several SMs. There is one single keyword to start the OTAP procedure. For parameters that are repeated in several SMs only the last value sent is valid. For example, an SM could look like this:

Install operation:
First SM:
OTAP_IMPNG
PWD:secret
JADURL:http://www.greatcompany.com/coolapps/mega.jad
APPDIR:a:/work/appdir
HTTPUSER:user
HTTPPWD:anothersecret
Second SM:
OTAP_IMPNG
PWD:secret
BEARER:gprs
APNORNUM:access.to-thenet.net
NETUSER:nobody
NETPWD:nothing
DNS:192.168.1.2
START:install

Delete operation:
OTAP_IMPNG
PWD:secret
APPDIR:a:/work/appdir
START:delete

The first line is required: it is used to identify an OTAP SM. All other lines are optional and their order is insignificant, each line is terminated with an LF: '\n' including the last one. The keywords, in capital letters, are case sensitive. A colon separates the keywords from their values.

The values of APPDIR, BEARER and START are used internally and must be lower case. The password (PWD) is case sensitive. The case sensitivity of the other parameter values depends on the server application or the network. It is likely that not all parameters can be sent in one SM. They can be distributed over several SMs. Every SM needs to contain the identifying first line (OTAP_IMPNG) and the PWD parameter if a mandatory password has been enabled. OTAP is started when the keyword START, possibly with a parameter, is contained in the SM and the parameter set is valid for the requested operation. It always ends with a reboot, either when the operation is completed, an error occurred, or the safety timer expired. This also means that all parameters previously set by SM are gone.

Apart from the first and the last line in this example, these are the parameters described in the previous section. Possible parameters for the START keyword are: "install", "delete" or nothing. In the last case, an install operation is done by default.

The network does not guarantee the order of SMs. So when using multiple SMs to start an OTAP operation their order on the receiving side might be different from the order in which they were sent. This could lead to trouble because the OTAP operation might start before all parameters are received. If you discover such problems, try waiting a few seconds between each SM.

8.5 Java File Format

In general, all Java files have to comply with the IMP-NG and ME specifications. There are certain components of the JAD file that the developer must pay attention to when using OTAP:

- MIDlet-Jar-URL: make sure that this parameter points to a location on the network where your latest JAR files will be located, e.g. http://192.168.1.3/datafiles/mytest.jar, not in the filesystem like file://a:/java/mytest/mytest.jar. Otherwise this JAD file is useless for OTAP.
- MIDlet-Install-Notify: this is an optional entry specifying a URL to which the result of an update/install operation is posted. That is the only way to get any feedback about the outcome of an install/update operation. The format of the posted URL complies with the IMP-NG OTA Provisioning specification. In contrast to the jar and jad file this URL must not be protected by basic authentication.
- MIDlet-Delete-Notify: this is an optional entry specifying a URL to which the result of a delete operation is posted. That is the only way to get any feedback about the outcome of a delete operation. The format of the posted URL complies with the IMP-NG OTA Provisioning specification. In contrast to the jar and jad file this URL must not be protected by basic authentication.
- MIDlet-Name, MIDlet-Version, MIDlet-Vendor: are mandatory entries in the JAD and Manifest file. Both files must contain equal values, otherwise result 905 (see Section 8.7) is returned.
- MIDlet-Jar-Size must contain the correct size of the jar file, otherwise result 904 (see Section 8.7) is returned.

Example:

```
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: TLR Inc.
MIDlet-Jar-URL: http://192.168.1.3/datafiles/MyTest.jar
MIDlet-Description: My very important test
MIDlet-1: MyTest, , example.mytest.MyTest
MIDlet-Jar-Size: 1442
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1
```

A suitable Manifest file for the JAD file above might look like:

```
Manifest-Version: 1.0
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: TLR Inc.
MIDlet-1: MyTest, , example.mytest.MyTest
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1
```

8.6 Procedures

8.6.1 Install/Update

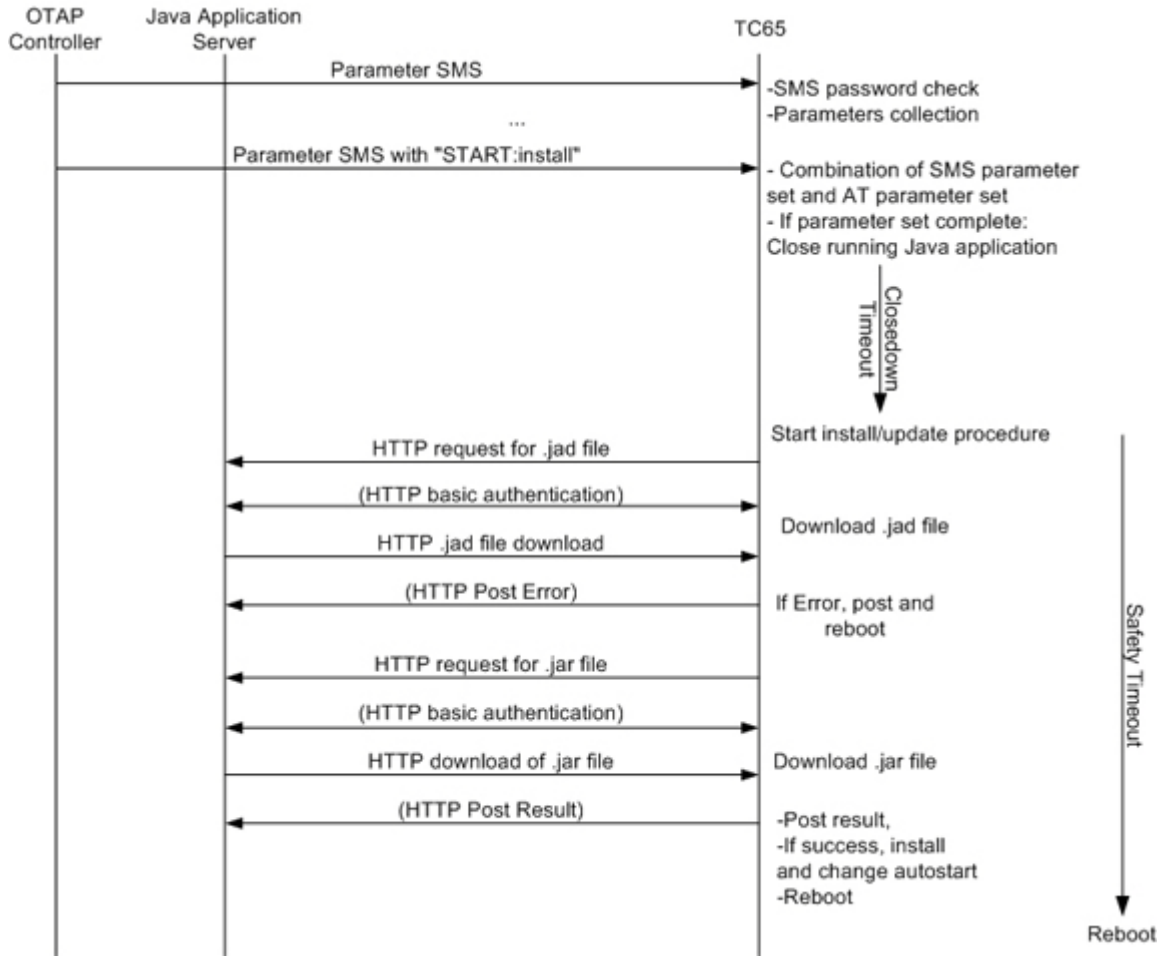


Figure 22: OTAP: Install/Update Information Flow (messages in brackets are optional)

When an SM with keyword START:install is received and there is a valid parameter set for the operation, the module always reboots either when the operation completed, an error occurred or the safety timer expired. If there is any error during an update operation the old application is kept untouched, with one exception. If there is not enough space in the file system to keep the old and the new application at the same time, the old application is deleted before the download of the new one, therefore it is lost when an error occurs. If install/update was successful, autostart is set to the new application.

8.6.2 Delete

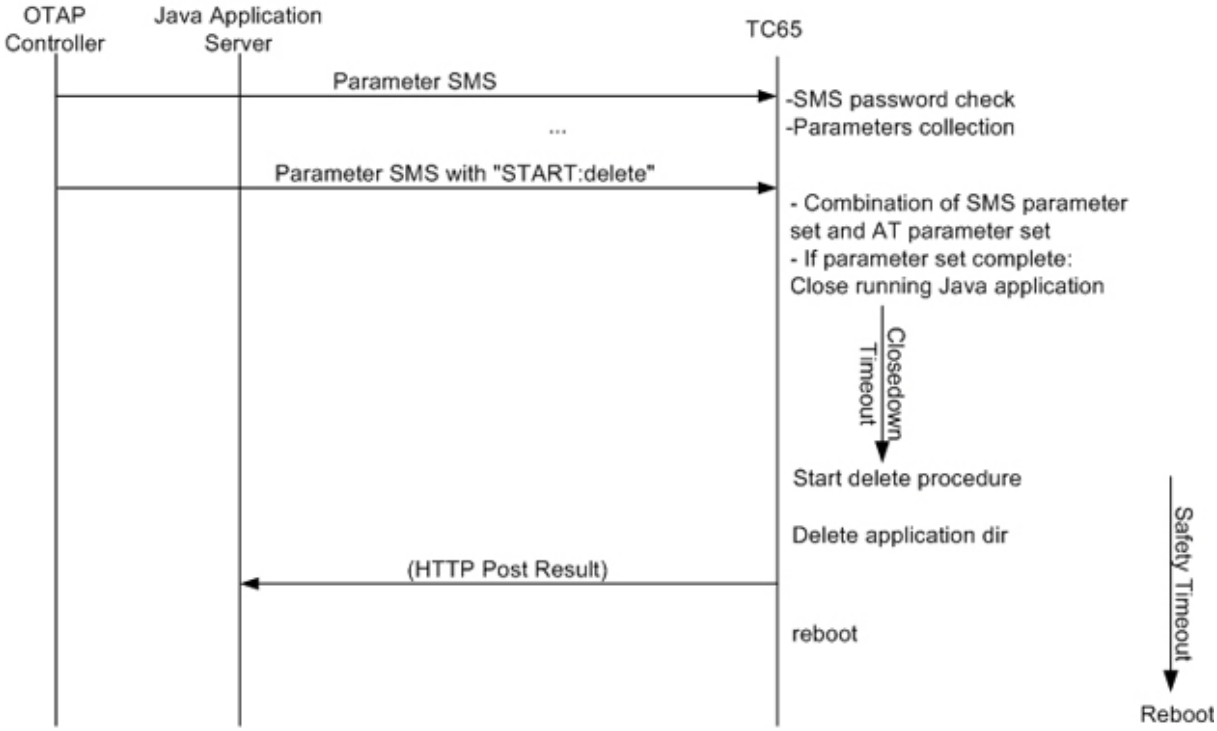


Figure 23: OTAP: Delete Information Flow (messages in brackets are optional)

When an SM with keyword START: delete is received and there is a valid parameter set for this operation, the module reboots either when the operation completed, an error occurred or the safety timer expired. If there is any error the application is kept untouched. Autostart is not changed.

8.7 Time Out Values and Result Codes

Timeouts:
- Closedown Timeout: 10 seconds
- Safety Timeout: 20 minutes

Result Codes: Supported status codes in body of the http POST request:
- 900 Success
- 901 Insufficient memory in filesystem
- 902 - not supported-
- 903 - not supported-
- 904 JAR size mismatch, given size in JAD file does not match real size of jar file
- 905 Attribute mismatch, one of the mandatory attributes MIDlet-name, MIDlet-version, MIDlet-Vendor in the JAD file does not match those given in the JAR manifest
- 906 Invalid descriptor, something is wrong with the format of the .jad file
- 907 invalid JAR, the JAR file was not available under MIDlet-Jar-URL, files could not be extracted from JAR archive, or something else is wrong with the format of the file.
- 908 incompatible configuration or profile
- 909 application authentication failure, signature did not match certificate
- 910 application authorization failure, tried to replace signed with unsigned version
- 911 -not supported-
- 912 Delete Notification

All HTTP packets (GET, POST) sent by the module contain the IMEI number in the User-Agent field, e.g.

User-Agent: <productname>/000012345678903 Profile/IMP-NG Configuration/CLDC-1.1

This eases device identification at the HTTP server.

8.8 Tips and Tricks for OTAP

- For security reasons it is recommended that an SMS password be used. Otherwise the 'delete' operation can remove entire directories without any authentication.
- For extra security, set up a private CSD/PPP Server and set its phone number as a fixed parameter. This way, applications can only be downloaded from one specific server.
- As a side effect, OTAP can be used to simply reboot the module. Simply start an OTAP procedure with a parameter set which will not really do anything, such as a delete operation on a nonexistent directory.
- If you do not want to start OTAP by SMS let your Java application do it by issuing the AT^SJOTAP command. This triggers an install/update operation as described in Section 8.6.1 but without the SMS part.
  Note: If a malfunctioning Java application is loaded the SM method will still be needed for another update.
- The OTAP procedure cannot be tested in the debug environment.
- Be aware that the module needs to be logged into the network to do OTAP. That means that either the Java application must enter the PIN or the PIN needs to be disabled.

- In some networking infrastructures there may be an interference between OTAP started on a circuit-switched bearer (CS) and an existing CS call. As a result, an OTAP failure may occur.
  If such a scenario is likely to happen in your application environment we recommend that a delay of around 15s be set between hanging up the CS call and starting OTAP over CSD. This means that you cannot use the standard SM trigger for OTAP because there are no 15s time when shutting down the running Java application. Instead, trigger OTAP from your application (e.g. receive trigger SM from application, hang up CS call, wait 15s, start OTAP by AT command).
  In emergency cases (when your Java application is malfunctioning) you can still use the SM trigger for OTAP but you probably need to try several times to make sure that there is no CS call at the time OTAP starts over CSD.

8.9 OTAP Tracer

For easy debugging of the OTAP scenario, the OTAP procedure can be traced over the serial interface. The trace output shows details of the OTAP procedure and the used parameters. To enable the OTAP trace output use the AT command AT^SCFG, e.g.
AT^SCFG=Trace/Syslog/OTAP,1

The serial interface on which you issue this command is then exclusively used for the OTAP tracer. All other functionality which is normally present (AT commands or CommConnection and System.out in Java) is not available when the tracer is on.

This feature is intended to be used during development phase and not in deployed devices.

8.10 Security

Java Security as described in Chapter 11 also has consequences for OTAP. If the module is in secured mode the MIDlet signature is also relevant to the OTAP procedure. This means:

- If the application is an unsigned version of an installed signed version of the same application then status code 910 is returned.
- If the applications signature does not match the module's certificate then status code 909 is returned.

8.11 How To

This chapter is a step-by-step guide for using OTAP.

1. Do you need OTAP? Is there any chance that it might be necessary to update the Java application, install a new one or delete it? It could be that device is in the field and you cannot or do not want to update it over the serial line. If the answer is "yes" then read through the following steps, if the answer is "no" then consider simply setting the OTAP SMS password to protect your system. Then you are finished with OTAP.
2. Take a look at the parameters (Section 8.3), which control OTAP. You need to decide which of them you want to allow to be changed over the air (by SMS) and which you do not. This is mainly a question of security and what can fit into a short message. Then set the "unchangeable" parameters with the AT command (AT^SJOTAP).

3. Prepare the http server. The server must be accessible from your device over TCP/IP. That means there is a route from your device over the air interface to the http server and back. When in doubt, write a small Java application using the httpConnection Interface to test it.
4. Prepare the JAR and JAD files which are to be loaded over the air. Make sure that these files conform to the requirements listed in Section 8.5 and that they represent a valid application which can be started by AT^SJRA.
5. Put the files (JAR and JAD) on the http Server. The files can either be publicly available or protected through basic authentication. When in doubt try to download the files from the server by using a common Web browser on a PC, which can reach your http server over TCP/IP.
6. Prepare the SM sender. The sender must be able to send SMs, which conform to Section 8.4, to your device. When in doubt try to send "normal" SMs to your device which can than be read out from the AT command interface.
7. Test with a local device. Send a suitable short message to your device, which completes the necessary parameter, sets and starts the operation. The operation is finished when the device reboots. You can now check the content of the file system and if the correct jar and jad files were loaded into the correct location.
8. Analyze errors. If the above test failed, looking at your device's behavior and your http servers access log can give you a hint as to what went wrong:
   - If the device did not terminate the running Java application and did not reboot, not even after the safety timeout, either your SM was not understood (probably in the wrong format) or it did not properly authenticate (probably used the wrong password) or your parameter set is incomplete for the requested operation.
   - If the device terminated the running Java application, but did not access your http server, and rebooted after the safety timeout, there were most likely some problems when opening the network connection. Check your network parameters.
   - If the device downloaded the jad and possibly even the jar file but then rebooted without saving them in the file system, most likely one of the errors outlined in Section 8.5 occurred. These are also the only errors which will return a response. They are posted to the http server if the jad file contains the required URL.
9. Start update of remote devices. If you were able to successfully update your local device, which is hopefully a mirror of all your remote devices, you can start the update of all other devices.

8.12 Incremental OTAP

As an extension of the standard OTAP procedure the Java Application can be provisioned in an incremental way to save some bandwidth when downloading a new version to a large number of deployed devices.

In order to make use of this feature the application has to be developed in a modular way. That means that the classes have to be split up into multiple jar files with corresponding jad files. There can be only one jar file containing the "midlet" class which is called the midlet, all the others are called liblets.

A liblets jar and jad file have to fullfil the same requirements as a midlets:
- The attributes name, version and vendor have to exist in both files with the same value
- The jar size needs to be present in the jad file and has to correspond with the jars file size
- The jar URL has to be present in the jad file and point to a valid location

The attribute names are slightly different.

Jad file example:

```
LIBlet-Name: network
LIBlet-Version: 2.0.0
LIBlet-Vendor: Customer Inc.
LIBlet-Jar-Size: 1259
LIBlet-Jar-URL: http://customer.server.com/otap/network.jar
```

Corresponding manifest file example:

```
LIBlet-Name: network
LIBlet-Version: 2.0.0
LIBlet-Vendor: Customer Inc.
```

If the liblet is signed the attributes
- MicroEdition-Configuration
- MicroEdition-Profile

also need to be corresponding in both files if they exist.

The jad file of the midlet needs to be extended to specify the liblets needed, e.g.

```
Dependency-1: at; Customer Inc. ; 1.1.2; liblet
LIBlet-Dependency-JAD-URL-1: http://customer.server.com/otap/at.jad
Dependency-2: network; Customer Inc. ; 2.0.0; liblet
LIBlet-Dependency-JAD-URL-2: http://customer.server.com/otap/network.jad
```

The numbering starts with 1 and has to be continuous. The "Dependency" attribute consists of name, vendor, version and "liblet"-keyword, all separated by semicolons.

The name is used to identify a liblet. It has to correspond with the name attribute of the liblet. Vendor and "liblet" entries are currently unused.

The OTAP install/update process still always starts with the download of the midlets jad file. After that only the parts which are necessary (i.e. version number in descriptor differ from installed version number) are downloaded and installed into the Application Directory.

PRE-deletion (i.e. remove an old jar file before download of the new one in order to make room in the ffs) of installed files is not supported for incremental OTAP.

There is only one result posted at the end of the whole OTAP process. The URL for that can (as usual) be specified by the otap at command or be included in a jad file. When multiple jad files involved in the process contain a NotifyURL the one from the file which was processed last is used. So you could e.g. put a different NotifyURL into each jad file, that will give you an extra hint at what point the OTAP process went wrong.

The name of the attribute is the same in all jad files: MIDlet-Install-Notify or MIDlet-Delete-Notify

9 Compile and Run a Program without a Java IDE

This chapter explains how to compile and run a Java application without a Java IDE.

9.1 Build Results

A JAR file must be created by compiling an CMTK project. A JAR file will contain the class files and auxiliary resources associated with an application. A JAD file contains information (file name, size, version, etc.) on the actual content of the associated JAR file. It must be written by the user. The JAR file has the ".jar" extension and the JAD file has the ".jad" extension. A JAD file is always required no matter whether the module is provisioned with the Module Exchange Suite, as described in Section 7.1, or with OTA provisioning. OTA provisioning is described in Chapter 7.

In addition to class and resource files, a JAR file contains a manifest file, which describes the contents of the JAR. The manifest has the name manifest.mf and is automatically stored in the JAR file itself. An IMP manifest file for:

```
package example.mytest;
public class MyTest extends MIDlet
```

includes at least:

```
Manifest-Version: 1.0
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: Test Inc.
MIDlet-1: MyTest, example.mytest.MyTest
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1
```

A JAD file must be written by the developer and must include at least:

```
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: Test Inc.
MIDlet-1: MyTest, example.mytest.MyTest
MIDlet-Jar-URL: http://192.168.1.3/datafiles/MyTest.jar
MIDlet-Jar-Size: 1408
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1
```

A detailed description of these attributes can be found in [4].

9.2 Compile

- Launch a Command Prompt. This can be done from the Programs menu or by typing "cmd" at the Run... prompt in the Start menu.
- Change to the directory where the code to be compiled is kept.
  Compile the program with the SDK. Examples of build batch files can be found in each sample program found in the samples directory "Documents and Settings\All Users\Cinterion ABC2 WTK Examples\WTKSamples" under Windows XP or "Users\Public\ Cinterion ABC2 WTK Examples\WTKSamples" under Windows Vista and above. The samples directory can be opened directly via the WTK start menu entry.
- If the compile was successful the program can be transferred to the module and executed as described in the following chapters.

The batch files for compiling the samples are using the system environment variables JAVA_HOME_ABC2 and WTK_HOME_ABC2. The first one points to the root directory of the installed JDK and the second one to the root directory of the Cinterion-CMTK-<productname>-IMPNG installation. The installation process sets these environment variables. A modification is usually not necessary.

9.3 Run on the Module with Manual Start

- Compile the application at the prompt as discussed in Section 9.2 or in an IDE.
- Transfer the .jar and .jad file from the development platform to the desired directory on the module using the Module Exchange Suite or OTA provisioning. Chapter 7 explains how to download your application to the module.
- Start a terminal program and connect to ASC0.
- The command *AT^SJRA* is used to start the application and is sent to the module via your terminal program. Either the application can be started by .jar or by .jad file.

Example:
In your terminal program enter: *AT^SJRA=a:/java/jam/example/helloworld/helloworld.jar*
If you prefer to start with .jad file: *AT^SJRA=a:/java/jam/example/helloworld/helloworld.jad*
The Flash file system on the module is referenced by "a:".

Depending on which file you specify the java application manager tries to find the corresponding file in the same directory. This search is not done by name, but by comparing the contained attributes. The first file which contains the same values for MIDlet-Name, MIDlet-Version and MIDlet-Vendor is used.

9.4 Run on the Module with Autostart

- Compile the application at the prompt as discussed in Section 9.2 or in an CMTK integrated IDE.
- Transfer the .jar and .jad file from the development platform to the desired directory on the module using the Module Exchange Suite or OTA provisioning. See Chapter 7.

9.4.1 Switch on Autostart

- There is an AT command, *AT^SCFG*, for configuring the autostart functionality. Please refer to [1].
- Restart the module.

9.4.2 Switch off Autostart

There are three methods for switching off the autostart feature:
- the AT^SCFG command, or
- the graphical "autoexec_off.exe" tool (included in the Installation CD software under wtk/bin), or
- the command line "cmd_autoexec_off.exe" tool (included in the Installation CD software under wtk/bin).

To disable the automatic start of a user application in a module these steps must be carried out:

Using the graphical "autoexec_off.exe" tool:
1. Connect the module to the PC
2. Make sure, that the module is switched off
3. Start the "Autoexec_Off" program
4. Select the COM-Port
5. Press the "AutoExec Off" button

Using the command line tool "cmd_autoexec_off.exe":
1. Connect the module to the PC
2. Make sure, that the module is switched off
3. Start the command line tool "cmd_autoexec_off" with option <com-port> (com1 up to com99 for com port selection)

10 Compile and Run a Program with a Java IDE

10.1 Debug Environment

10.1.1 Data Flow of a Java Application in the Debug Environment

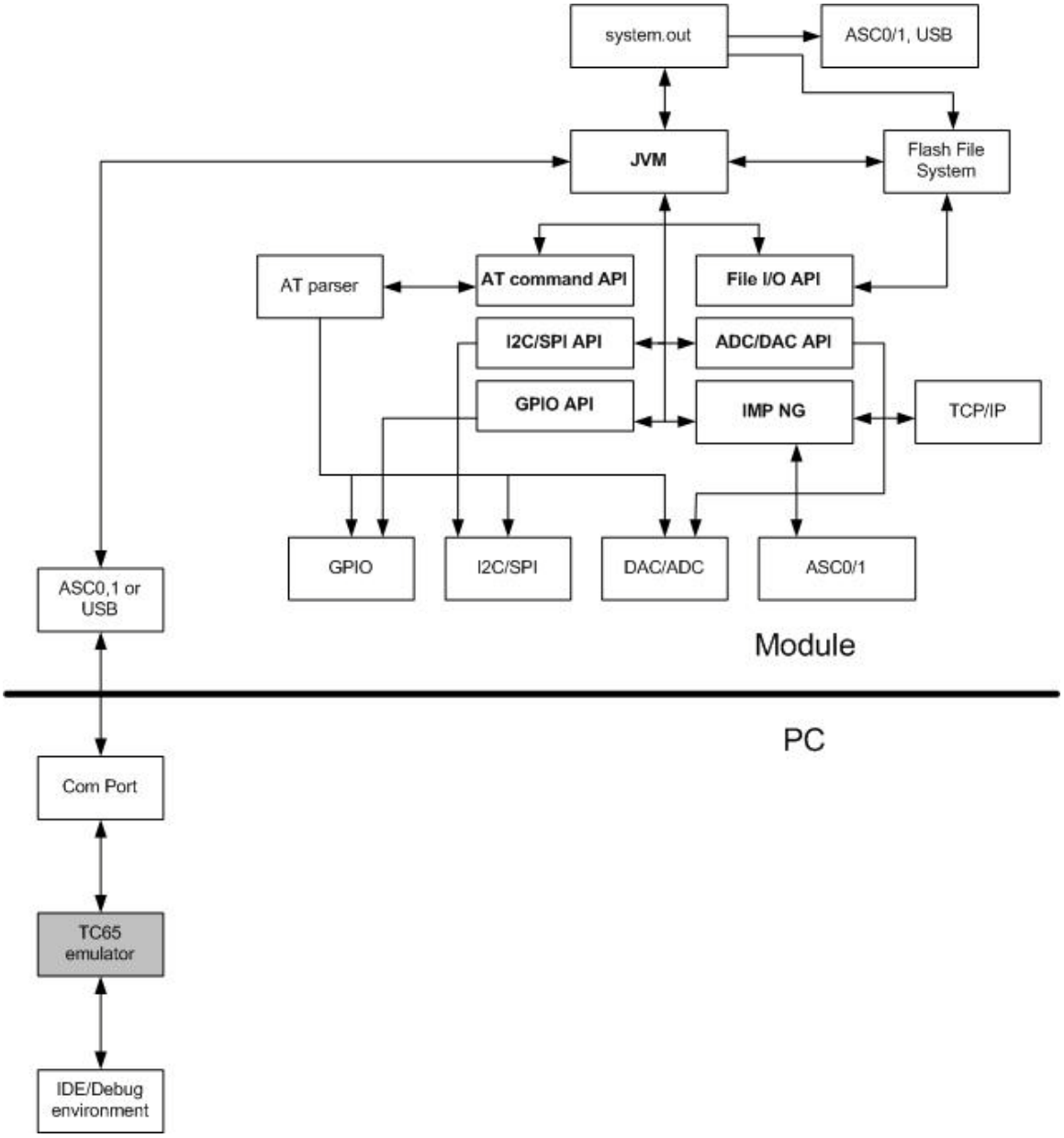


Figure 24: Data flow of a Java application in the debug environment

In the debug environment the module is connected to a PC via a serial interface. This can be a USB or RS232 line. The application can then be edited, built, debugged or run within an IDE on the PC. When running or debugging the MIDlet under IDE control it is executed on the module (on-device execution) and not on the PC. This can be either debugging mode, where the MIDlet execution can still be controlled from the IDE (on-device debugging) or normal mode, where the MIDlet is copied to the module and started normally. This ensures that all interfaces behave the same whether debugging mode is used or not.

10.1.2 Emulator

The ME emulator is part of the CMTK and is used as the controlling entity for on-device debugging. Some values can be configured in the file "wtk/bin/WM_Debug_config.ini". The emulator runs fine without changes in WM_Debug_config.ini file, because the settings for the serial interface (COM port and baudrate) are configured during installation of CMTK.

Debugging information between the Debugger (IDE) and the JVM is transferred over an IP connection. In order to establish this IP connection between the PC and the module the emulator needs a special Dial-Up-Network (DUN):

- ISP name: "IP connection for remote debugging of ABC2"
- Modems: "Cinterion ABC2 Java Debug Modem USB" for USB serial modem and "Cinterion ABC2 Java Debug Modem Ser" for standard serial modem
- Phone number: *88#
- Disable the Redial if line dropped option.
- Enable Connect automatically

This Dial-Up Network (DUN) connection is installed automatically together with the required modem device during installation of the CMTK. The emulator uses always the serial port configured for this Dial-Up Network connection.

You can use any of the three serial interfaces (ASC0, ASC1, USB) to connect with module, but you will lose the functionality which is normally present on the interface. Please also note that the Dial-Up Network under Windows needs the DCD line of the serial interface. The DCD line is not served on the ASC1 interface for the current Java modules. So you have to electrically set the DCD line to high for debugging over ASC1. This cannot be done by AT comand. In addition, the missing DTR input on ASC1 may prevent the module from detecting the termination of the Dial-Up Network, so in some cases the module has to be reset manually after the debug session was terminated. Because of the above mentioned issues and for performance reasons it is recommended to use the USB interface (see also Section 10.1.3).

There are some points to notice in case of using the USB interface in a debugging session. If you are starting a debugging session very quickly once again after end of a previous debugging session, the emulator is not able to select the USB interface and is selecting a standard modem connection. Please wait some seconds to avoid this operation system dependent problem, because the Windows operation system needs some seconds to enable the USB port once again after the "IP connection for remote debugging of ABC2" is closed.

If necessary, the IP addresses used for the debug connection can also be changed. This is done in the file "WM_Debug_config.ini". For details, see also the AT^SCFG command and its "Userware/DebugInterface" parameters described in [1]. Please keep in mind that the IP address range 10.x.x.x is not supported for in device debugging!

During installation of CMTK some new programs are installed for handling the debugging session in conjunction with the IDE. The installation routine of the CMTK doesn't change any configuration of an existing firewall on your PC. In the case, that a firewall is installed on your PC and the local configured and used IP connection (Dial-Up Network connection for debugging) is blocked or disturbed by this firewall, please configure the firewall or the Dial-Up Network connection manually to accept the new installed programs and the port or to use another port or contact your local PC administrator for help.

10.1.3 Change Baud Rate

By default, the Cinterion CMTK installation process implements the connection necessary for on device debugging with the maximum possible baud rate supported by both the target computer and the connected module. For optimized performance Cinterion recommends to use the USB interface for debug connections, because then the baud rate used on that virtual USB COM port is irrelevant.

If a serial COM port is employed however, the configured baud rate might not work reliably, depending on the quality of the used serial cable and COM port hardware. In these cases it might become necessary to change the baud rate of the debug connection manually. This has to be done as follows:

- First, the baud rate of the modem device "Cinterion ABC2 Java Debug Modem Ser" has to be set to the required maximum port speed. The modem properties are available via the Windows device manager or the phone and modem options (see Figure 25).

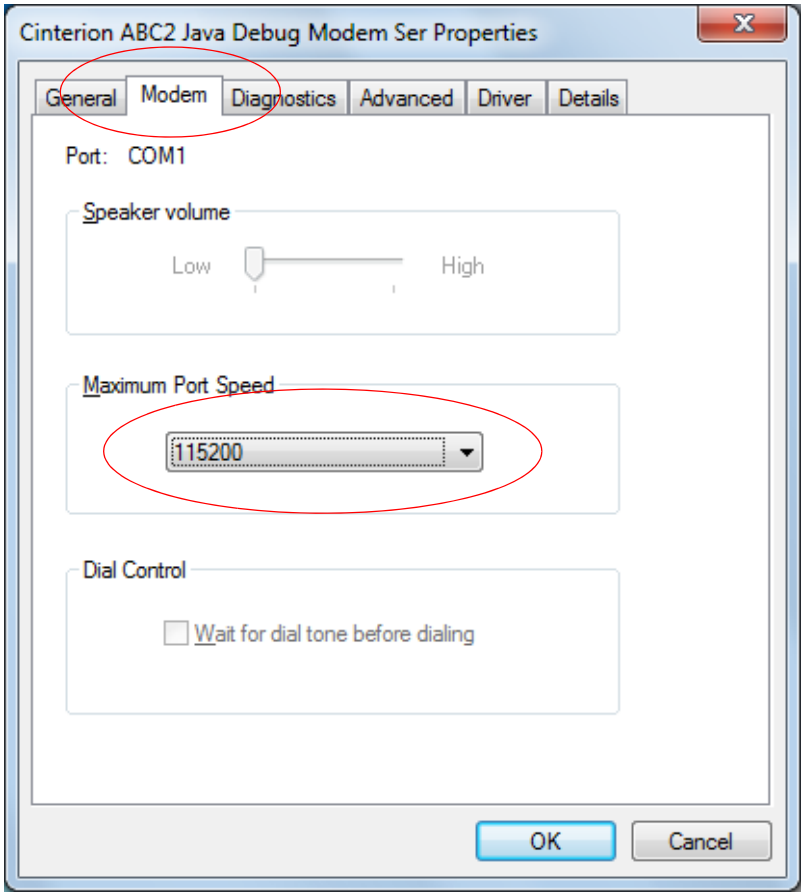


Figure 25: Specify maximum port speed for modem device

- Then the baud rate used by the debug connection "IP connection for remote debugging of ABC2" must be set to exactly the same value. The properties of the debug connection are available via the Windows network connection settings (see Figure 26 and Figure 27).

Please note that on device debugging does not work, if the maximum baud rates specified for modem device and debug connection are not identical. In such a case the emulator aborts debugging with an appropriate error message.

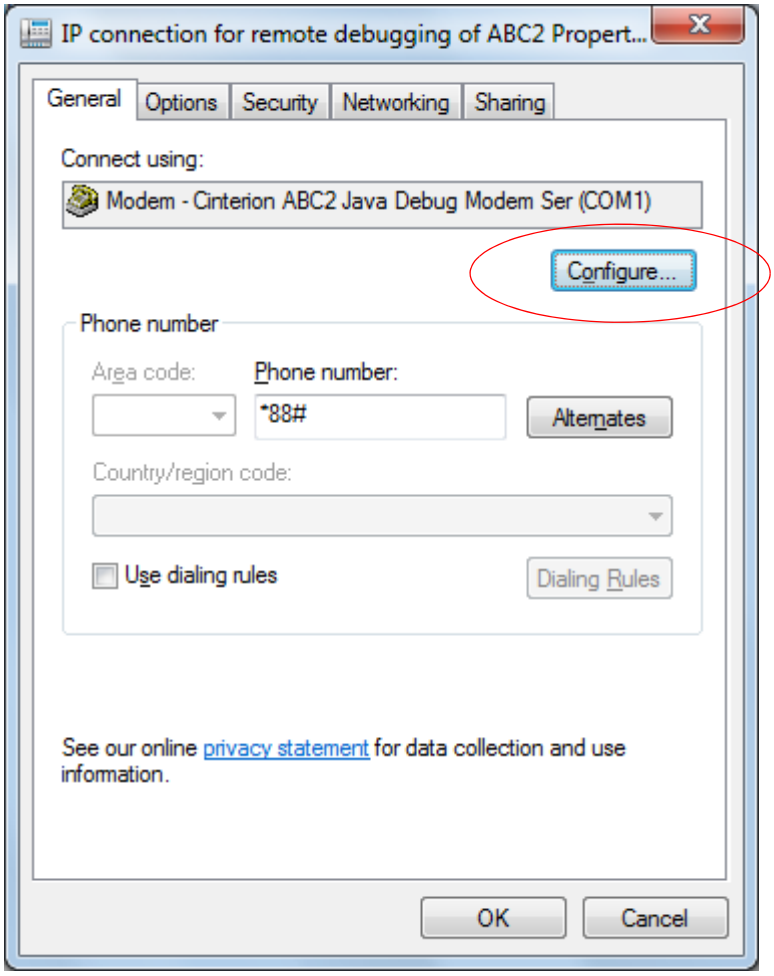


Figure 26: Configure debug connection

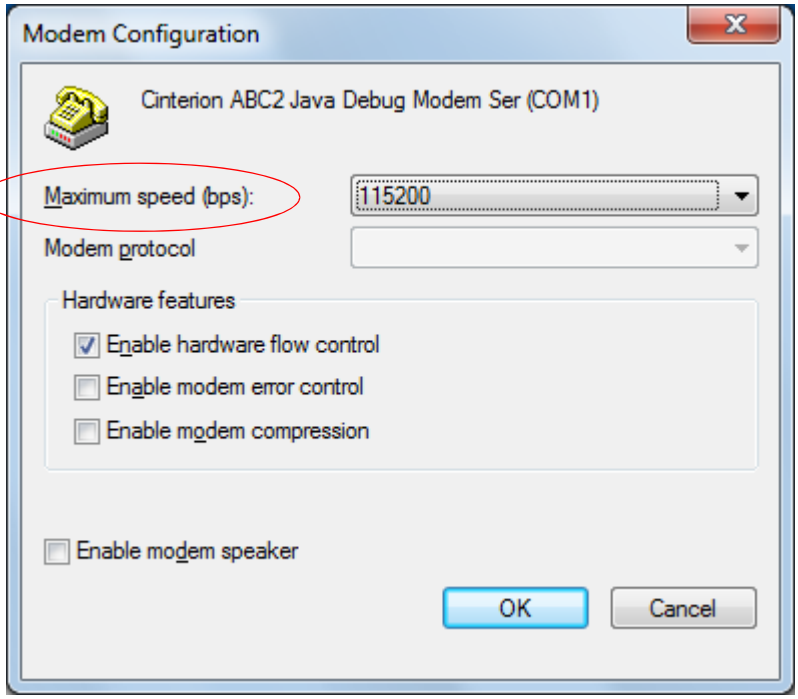


Figure 27: Specify baud rate for debug connection

10.2 Using Eclipse for Java Development

The Cinterion WTK is integrated automatically into the Eclipse IDE as of version 3.6.0 named "Helios" by the setup process if an appropriate Eclipse installation is found on the target computer. For usage of the Cinterion WTK inside Eclipse the "Mobile Tools for Java" (MTJ) plugin must be installed. Refer to Section 10.2.1 for information how to install this plug in.

In case there is no usable IDE found on the target computer the setup process offers the automatic installation of the Eclipse Helios Pulsar platform. This platform is preconfigured for mobile development and contains already the required MTJ plug in so that there are no further steps required. Please note that the Eclipse installation is not done via the Windows installer and therefore the installed files must be removed manually for a deinstallation. The setup process offers the possibility to enter the target path for the eclipse installation which is by default the all users profile.

10.2.1 Installing the "Mobile Tools for Java" Plugin

If there is already an appropriate Eclipse installation on the target computer the required MTJ plugin can be installed manually for usage of the Cinterion WTK via the menu item Help-->"Install New Software…". In the opened window select the download source "Helios - http://download.eclipse.org/releases/helios". After the list with available software has been obtained from the web site expand the list item "Mobile and Device Development" by clicking the small arrow on its left side and select the sub list item "Mobile Tools for Java". To install the selected plugin click Next and follow the offered steps.

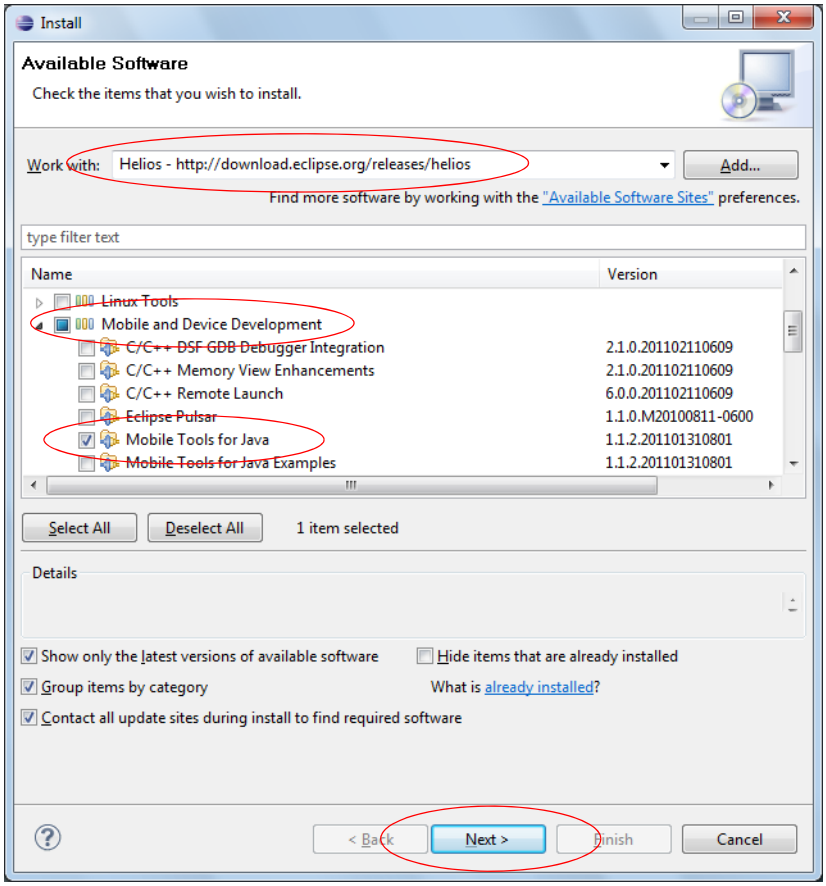


Figure 28: Installing the "Mobile Tools for Java" plugin

10.2.2 Integrating Cinterion WTK Manually

The integration of the Cinterion WTK into an appropriate Eclipse installation with added MTJ plugin can be done by the setup program automatically during first installation or afterwards in maintenance mode. Nevertheless, it can be done although manually which is described in this chapter. To add the WTK open the Eclipse menu item Window-->Preferences. In the opened window expand the preference list item "Java ME" and select the sub list item "Device Management". Then a list with the already installed Java ME devices is shown which is empty in the following example. For a manual installation select the button "Manual Install…"

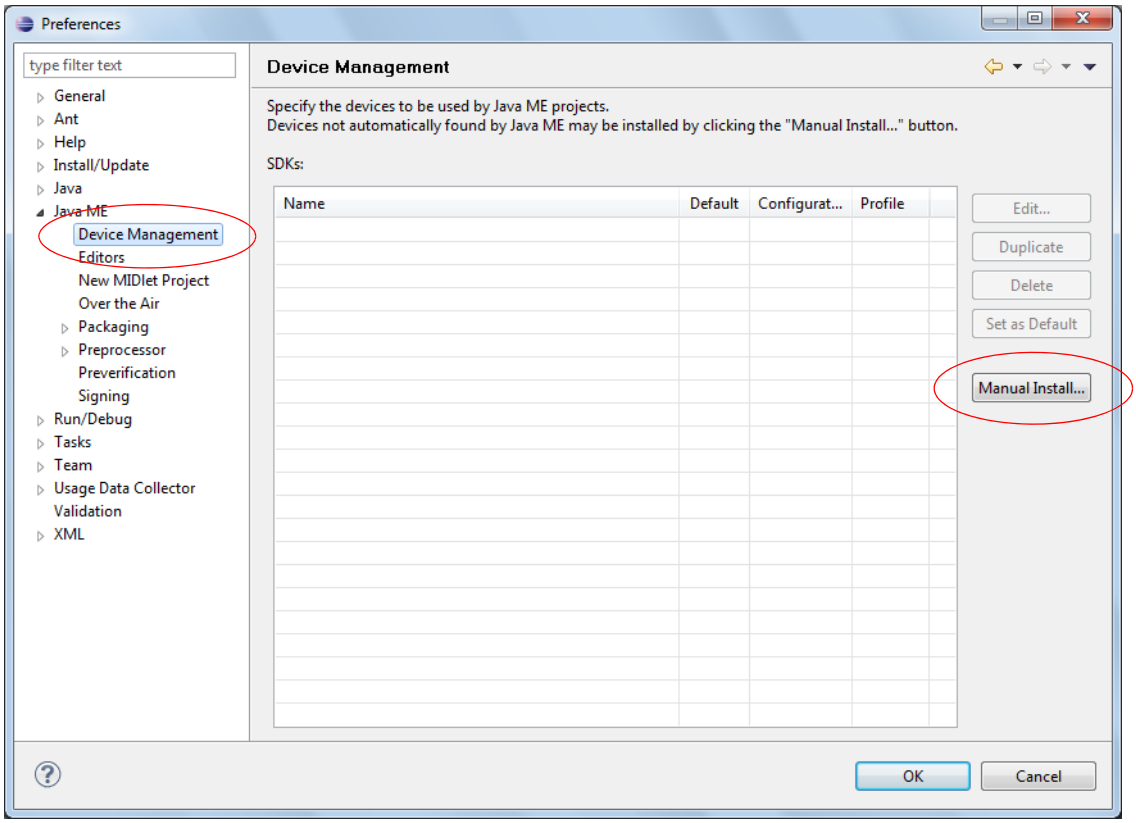


Figure 29: Integrating Cinterion WTK manually - Select preference

In the following window select the Browse…button, browse to the root folder of the Cinterion ABC2 WTK directory or enter its path "Program Files\Cinterion\CMTK\ABC2\WTK" directly. After the path has been entered it is scanned for usable CLDC device configurations and the found Cinterion WTK is added to the list. By clicking the Finish button it is added to the Eclipse IDE.

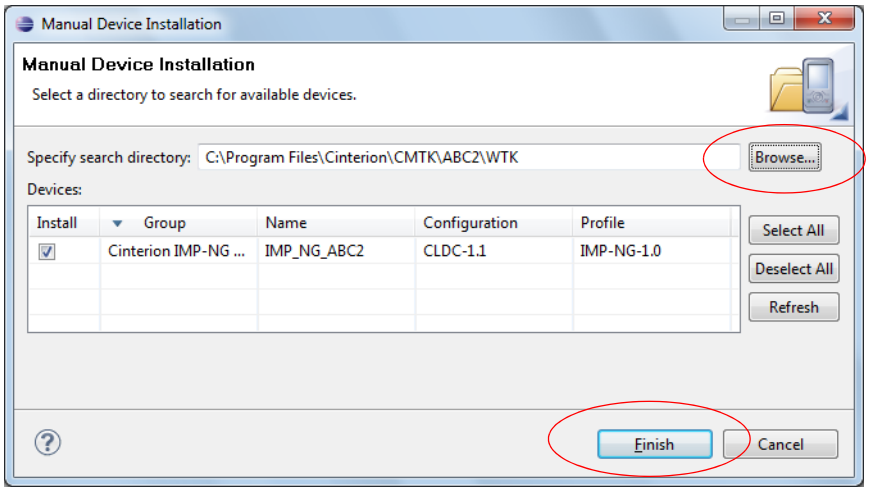


Figure 30: Integrating Cinterion WTK manually - Browse

Now the Cinterion WTK is available in the list of installed Java ME SDKs. To have the Cinterion WTK Java documentation directly available in your MIDlet projects the following additional steps are necessary. Select the new installed IMP_NG2_ABC2 device and click the Edit… button.

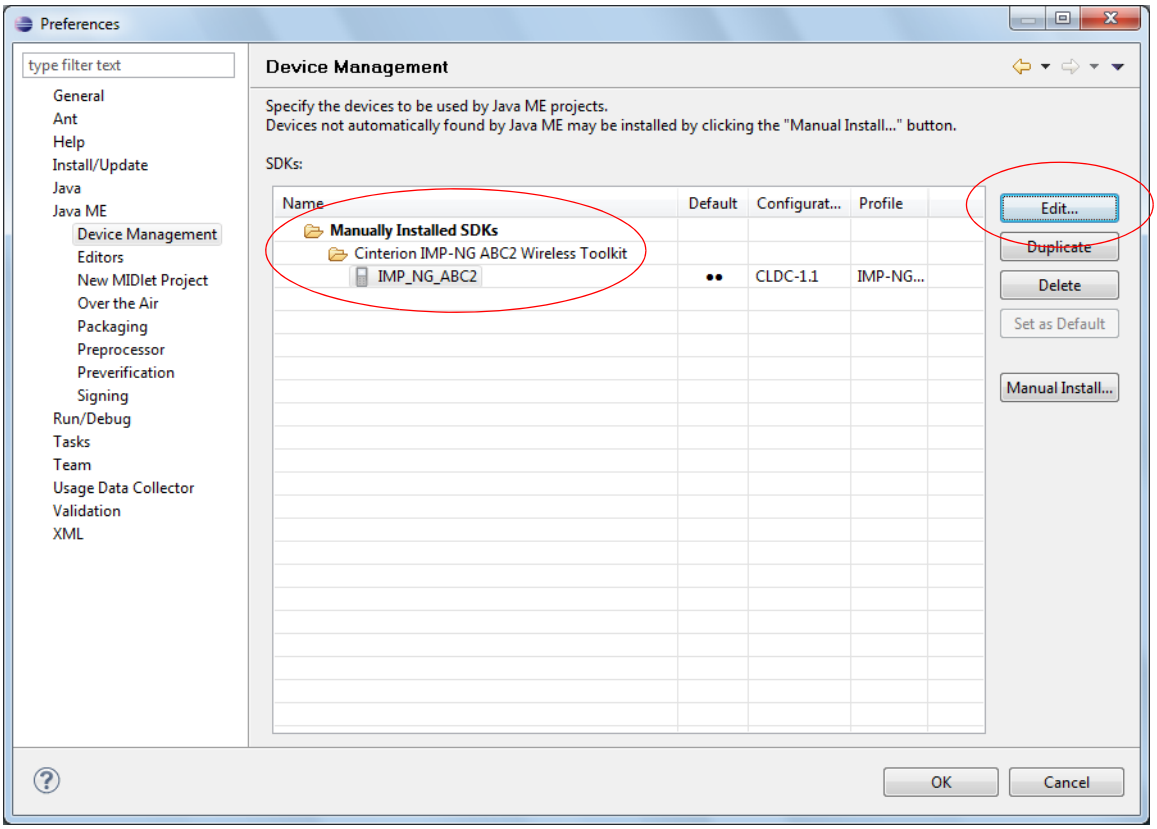


Figure 31: Integrating Cinterion WTK manually - Edit

In the following window select the Libraries pane and click into the Javadoc column. Once the table field is activated a small button with ellipsis appears which provides the possibility to browse to the WTK documentation folder.

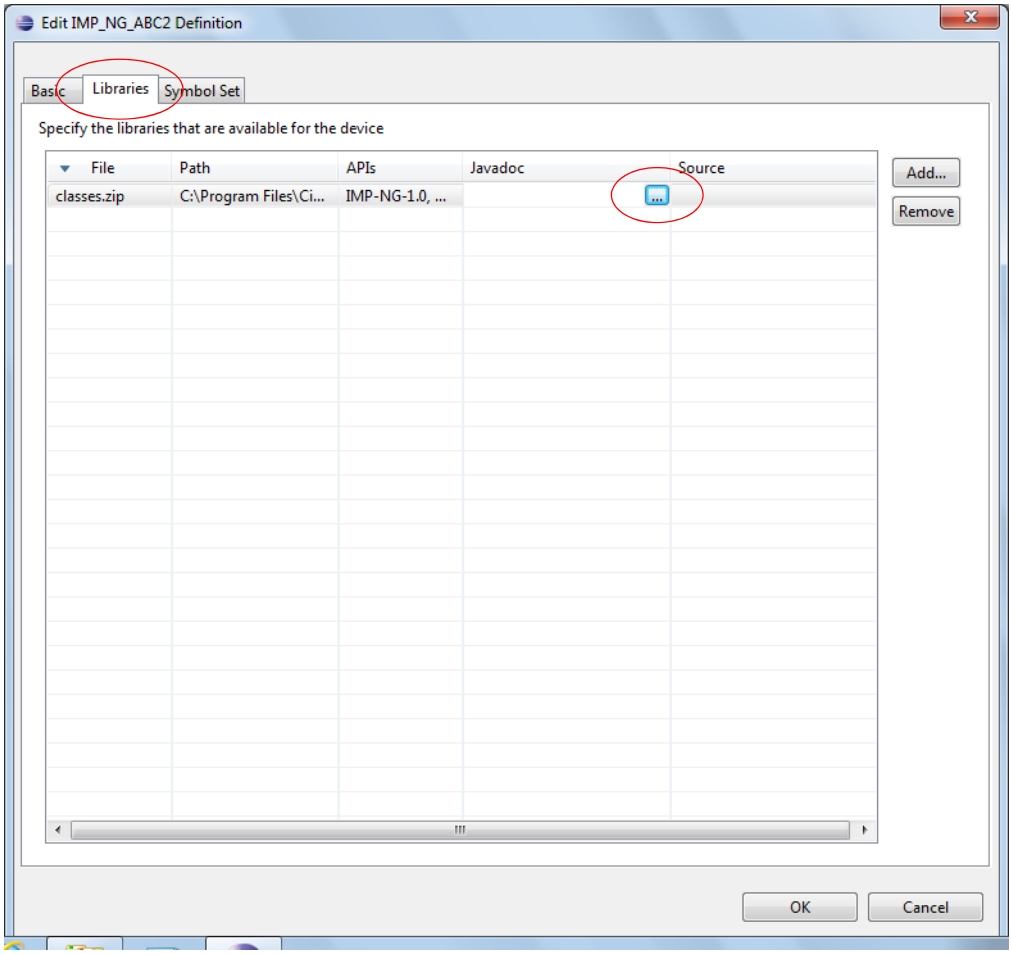


Figure 32: Integrating Cinterion WTK manually - Library

In the following window select "Javadoc URL" and navigate to the documentation folder which is "Program Files\Cinterion\CMTK\ABC2\WTK\doc\html" after clicking the Browse… button. The OK button then adds the Cinterion WTK documentation to the new Java ME device.

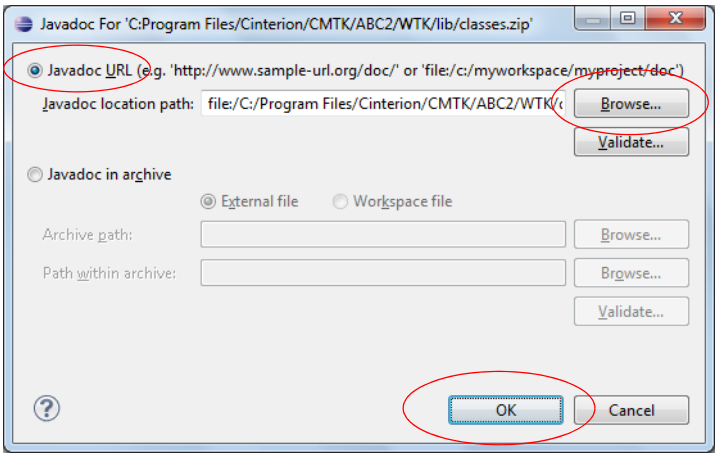


Figure 33: Integrating Cinterion WTK manually - Add WTK documentation

10.2.3 Import the provided WTK Samples

The Cinterion WTK provides the existing samples in an Eclipse project format as well. To import them into an Eclipse workspace select the Eclipse menu item File-->Import… In the following window expand the list item General and select the sub list item "Existing Projects into Workspace". After clicking the Next button it is possible to navigate to the folder containing the Cinterion WTK Eclipse samples called "Documents and Settings\All Users\Cinterion ABC2 WTK Examples\EclipseSamples" under Windows XP or "Users\Public\ Cinterion ABC2 WTK Examples\EclipseSamples" under Windows Vista and above.

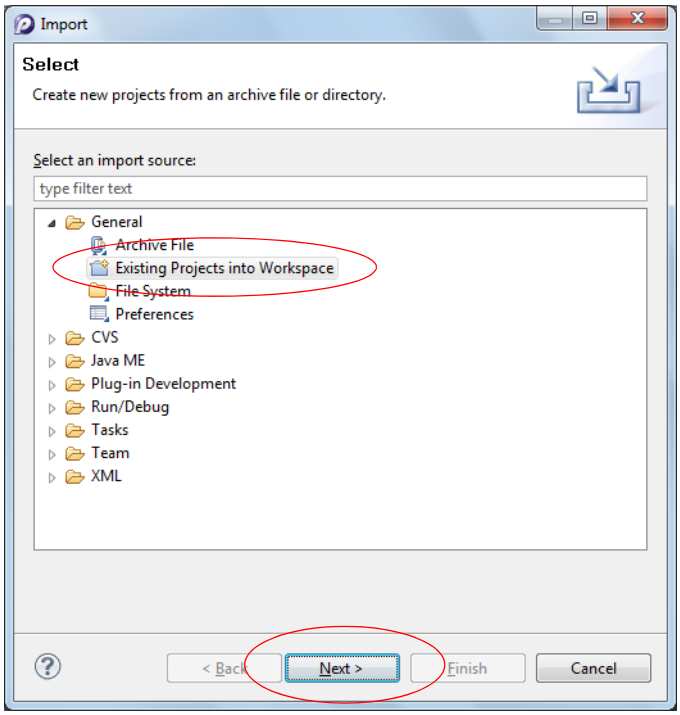


Figure 34: Import the provided WTK Samples - Select

Once the root directory of the samples has been entered the existing projects are added to the list and can be separately selected for import. Checking or unchecking the checkbox "Copy projects into workspace" controls whether the samples are copied into the current Eclipse workspace or kept where they are. After clicking the Finish button the selected samples are available for editing and debugging.

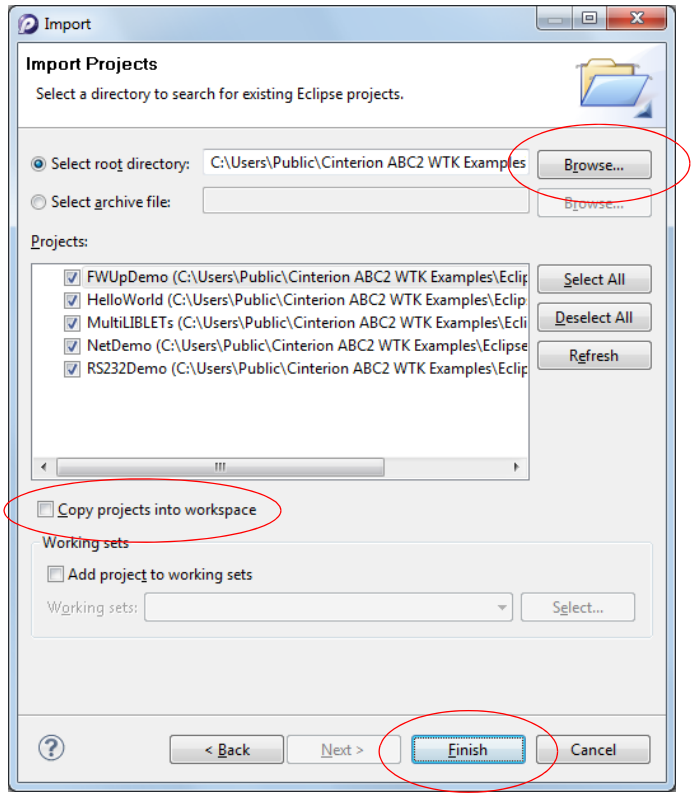


Figure 35: Import the provided WTK Samples - Copy

10.2.4 Creating a new MIDlet

To create a new MIDlet select the menu item File-->New-->Project…. In the following window expand the list item "Java ME" and select the sub list item "MIDlet Project".

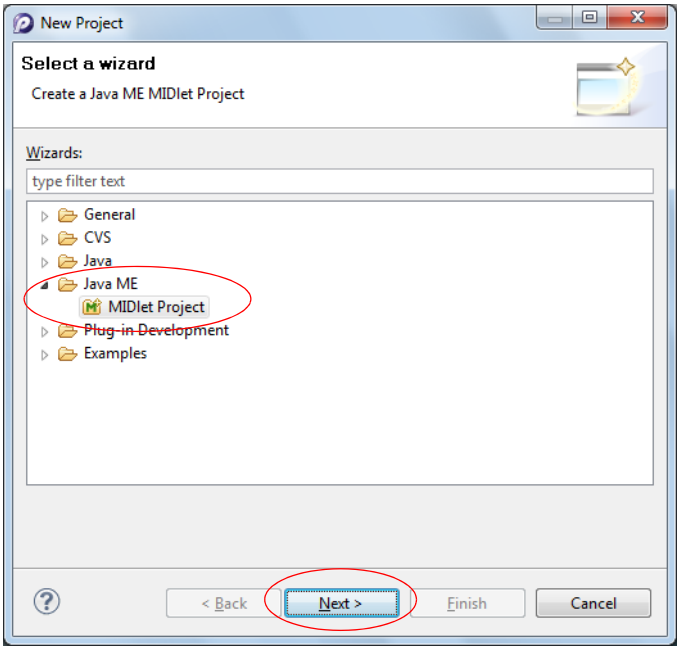


Figure 36: Creating a new MIDlet - Select wizard

After clicking the Next button a wizard for MIDlet creation appears. Specify the appropriate options as required for the new project. Ensure that only the "IMP_NG_ABC2" configuration is selected in the configuration list as active.

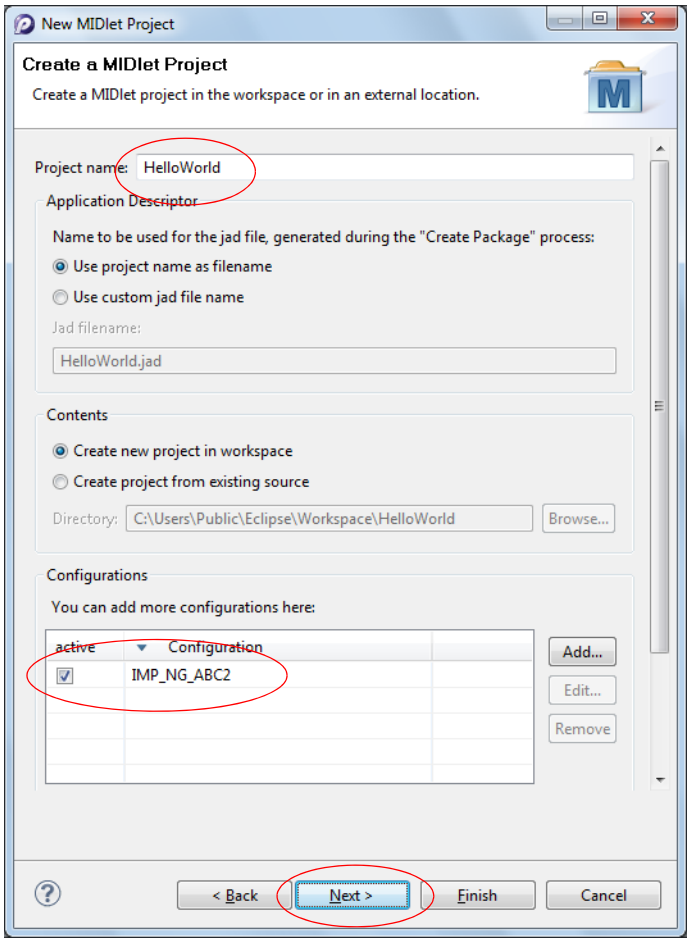


Figure 37: Creating a new MIDlet - Create project

After clicking the Next button a window for configuring the MIDlet project content opens. Ensure that the "Microedition Configuration" is set to "Connected Limited Device Configuration (1.1)" and the "Microedition Profile" is set to "Information Module Profile (NG)". After all necessary modifications have been done the project can be created by clicking the Finish button.

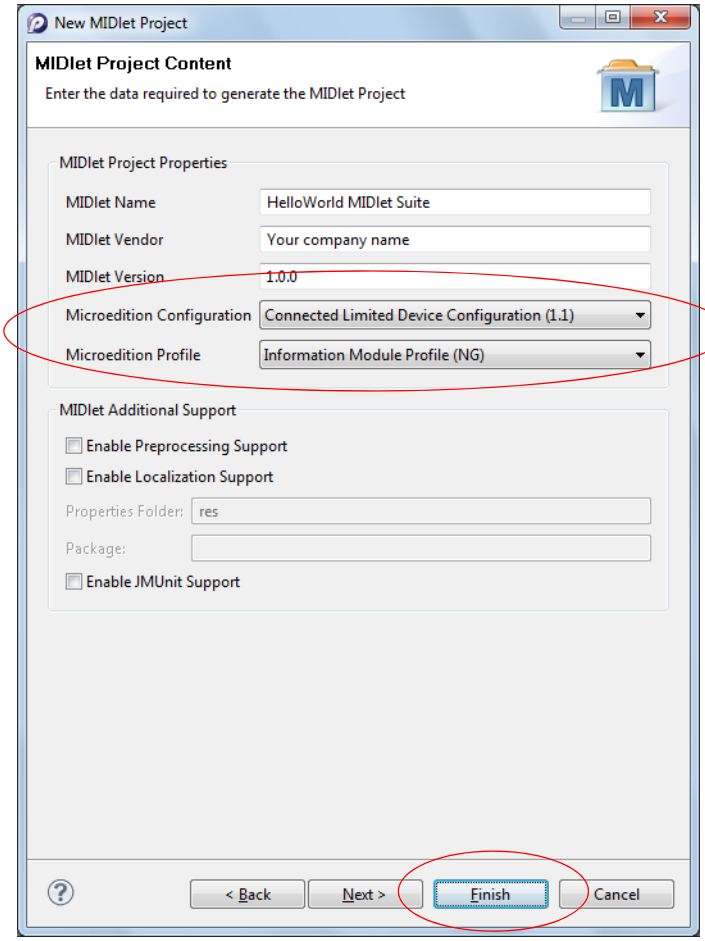


Figure 38: Creating a new MIDlet - Configure project

Now a newly created MIDlet project is available in the workspace.

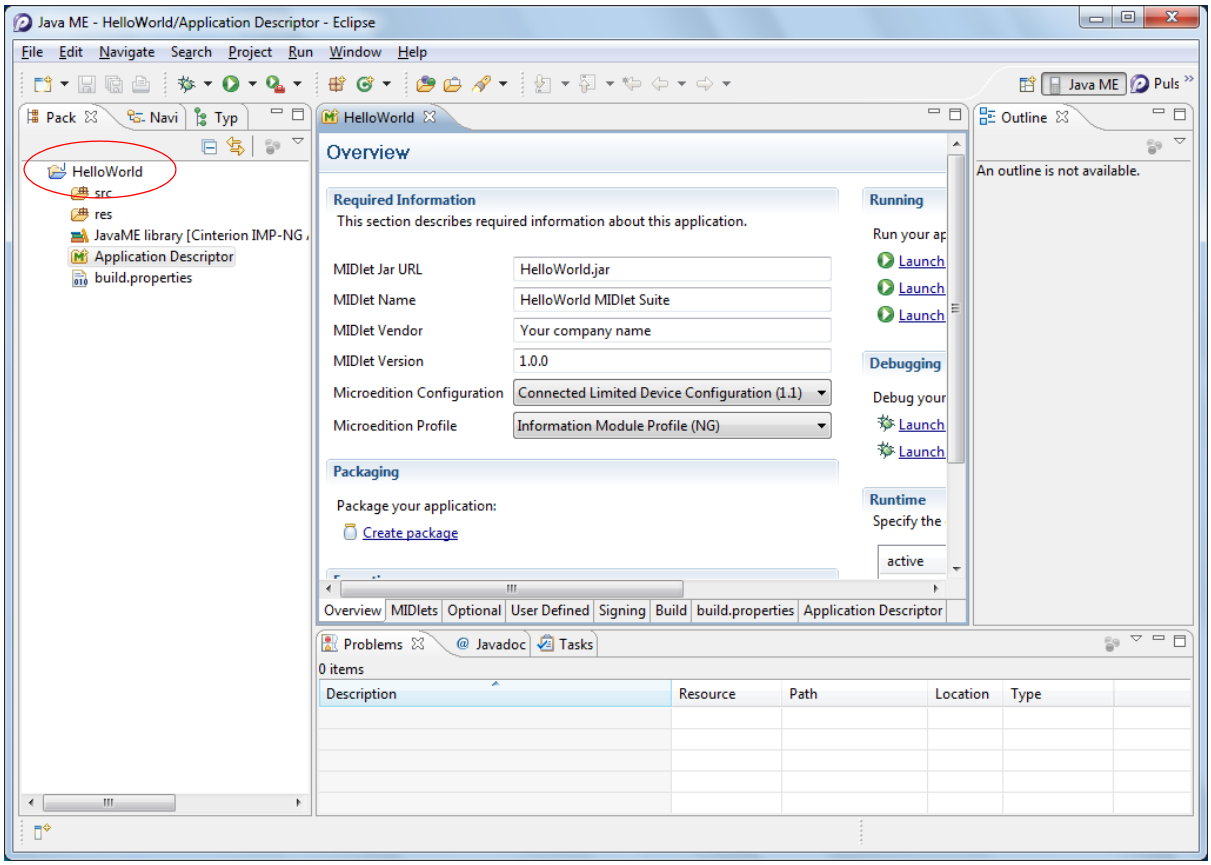


Figure 39: Creating a new MIDlet - Project overview

To ensure the correct Java compiler setting right click on the newly created project in the package tree explorer on the left side of the Eclipse window and select "Properties". In the opened project properties window select the item "Java Compiler" in the list on the left side. Ensure that the check box "Enable project specific settings" is selected and the "Compiler compliance level" is set to 1.3 or 1.4.

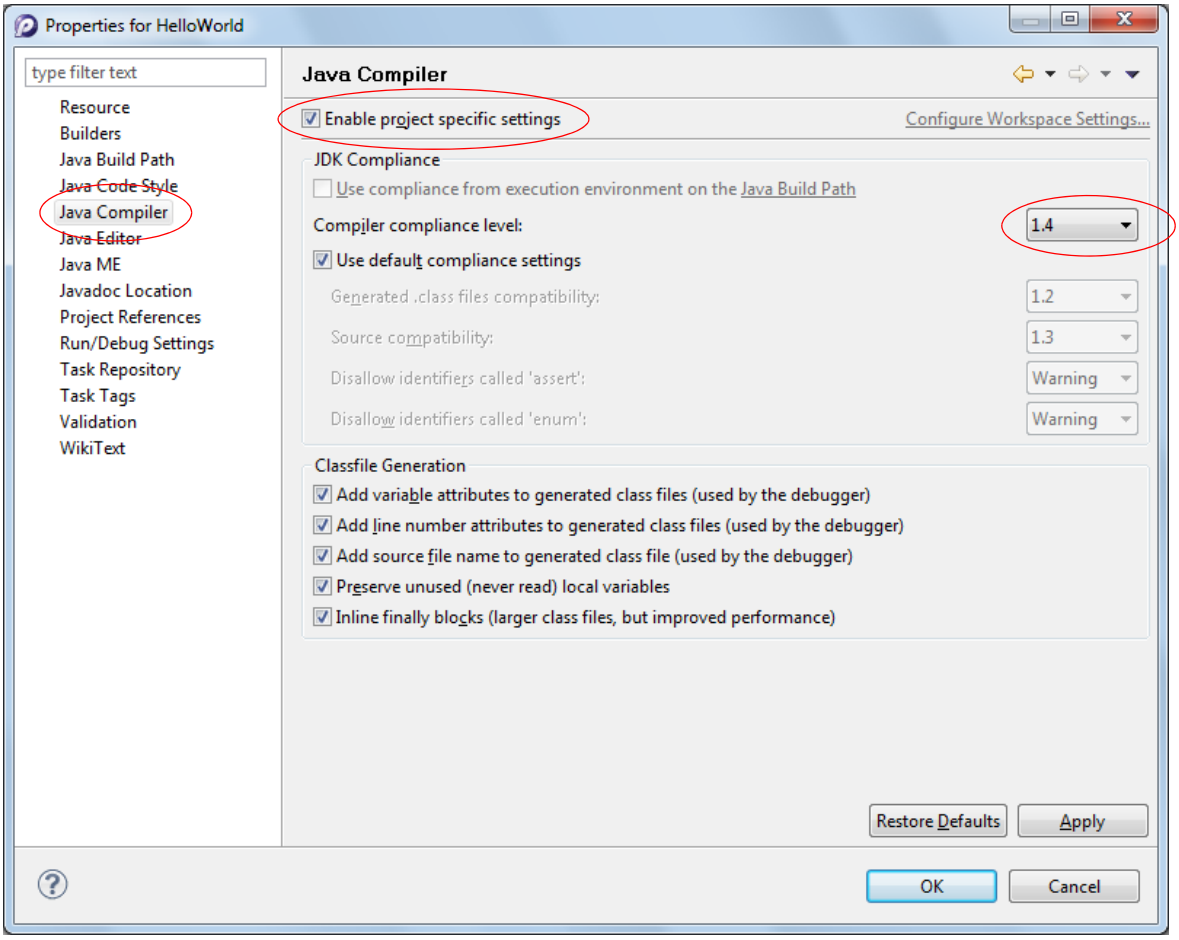


Figure 40: Creating a new MIDlet - Configure compliance level

To create the frame for a MIDlet program right click into the project tree and select New-->"Java ME MIDlet". In the following wizard enter the name for the MIDlet and click the Finish button.

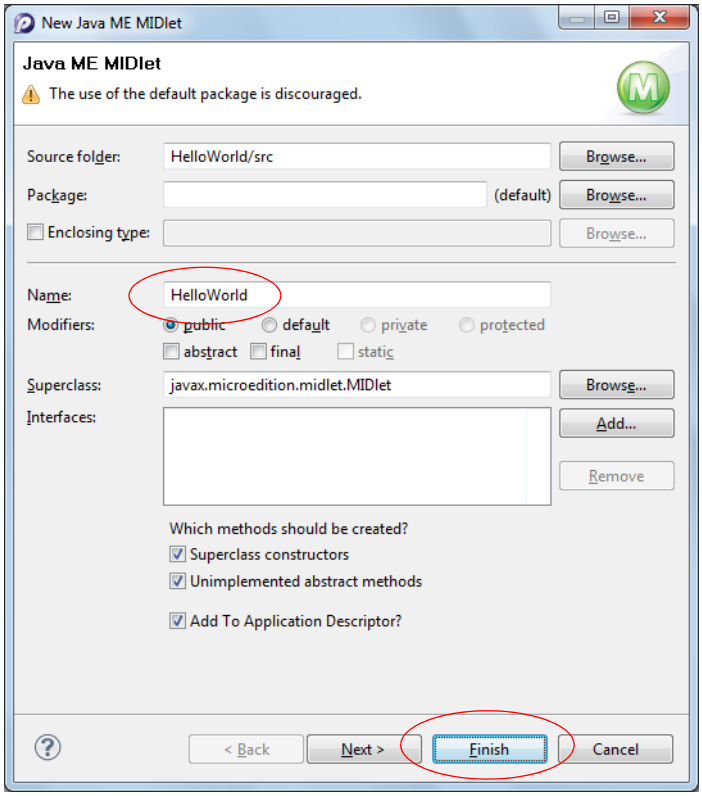


Figure 41: Creating a new MIDlet - Program name

Now, a corresponding Java file with the source of the basic MIDlet interface is created inside the source tree of the project. The only thing that remains to be done is to add a call of "notifyDestroyed()" to the "destroyApp()" method.

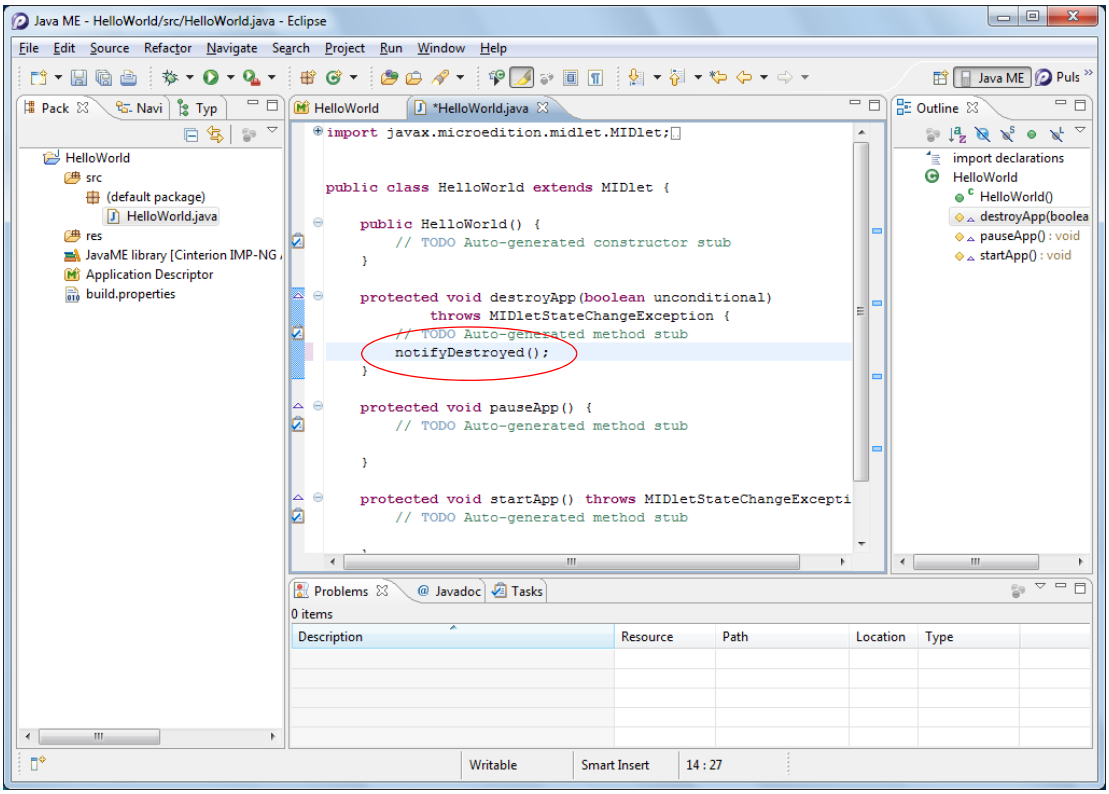


Figure 42: Creating a new MIDlet - HelloWorld.java

Finally, the actual functionality can be implemented and the project can be started and debugged inside the module as with other Eclipse projects. Please note that you must allow Eclipse and the debug agent to pass the firewall and configure the debug IP connection as a home network to be able to execute and debug MIDlets inside the device.

10.2.5 Using Eclipse Workspaces

Eclipse follows the concept of different "workspaces". Basically, these are folders on the hard disk used by Eclipse to store settings and new projects - just as an operating system might store documents and settings of different users in different directory structures. Eclipse can manage different workspaces allowing for an easy switching between different development environments requiring different IDE setups. If not disabled, Eclipse asks before each start for the workspace to be used.

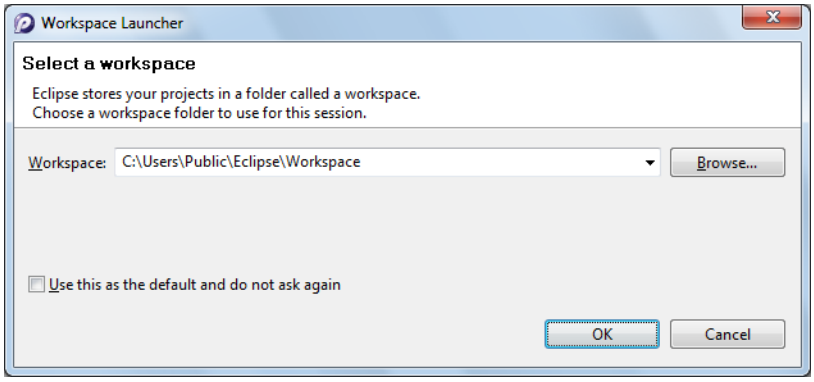


Figure 43: Using Eclipse workspaces

If Eclipse is installed via the Cinterion WTK setup a default workspace is created automatically with all required settings. But because Eclipse stores all settings inside the workspaces a new workspace which has been created via Eclipse it doesn't contain the necessary settings for using the Cinterion WTK. In this case the required settings can be added manually as described in the Section 10.2.2. Alternatively the Cinterion WTK setup can be executed in maintenance mode. After scanning for supported IDEs the required settings will be added to all available workspaces of a found Eclipse installation automatically.

10.3 Using NetBeans for Java Development

If an appropriate NetBeans installation (as of version 6.5 - 6.9.1) is found on the target computer, the Cinterion WTK is integrated automatically into the NetBeans IDE by the setup process. NetBeans 6.9.1 is distributed as part of the Cinterion installation CD and may be installed from there, if required. The package is located under the "Contribution" directory[2].

For usage of the Cinterion WTK inside NetBeans the "Mobility" plugin must be installed. Refer to Section 10.3.1 for information on how to install this plugin.

10.3.1 Installing the "Mobility" Plugin

The required "Mobility" plugin can be easily installed via the NetBeans plugin manager. This can be called via the menu item Tools-->Plugins. Inside the plugin manager activate the pane "Available Plugins" and inside the plugin list select the item "Mobility" in the category "Java ME". To install the selected plugin click the Install button and follow the offered steps.

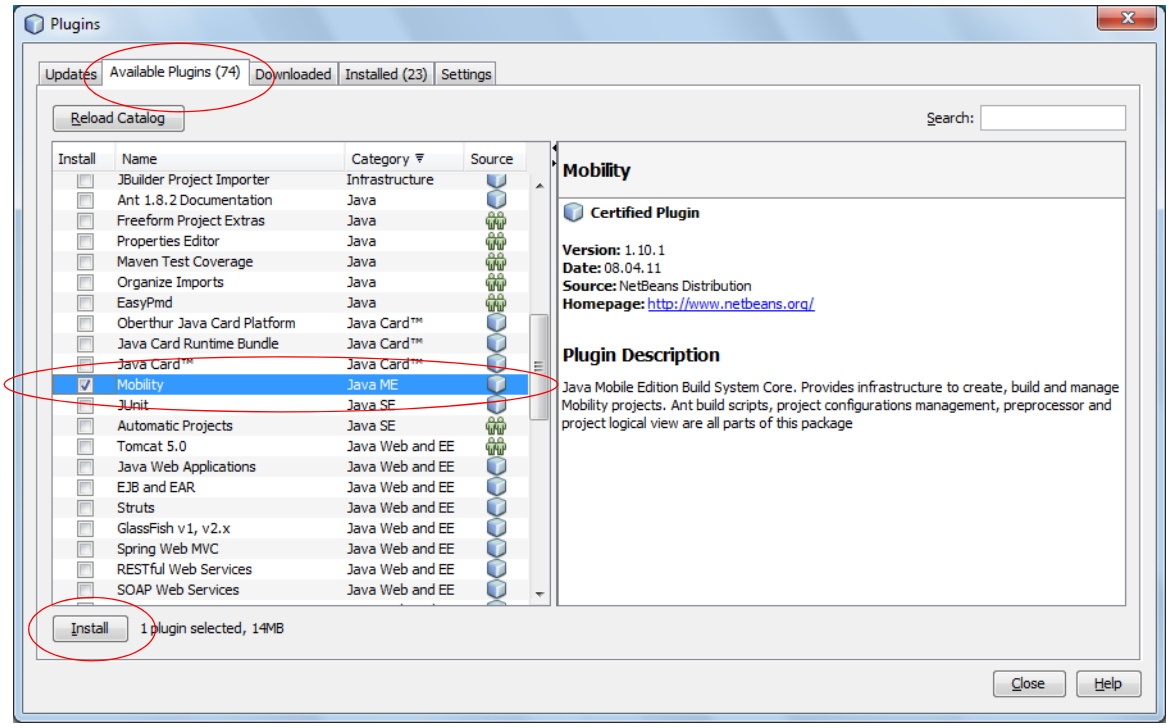


Figure 44: Installing "Mobility" plugin

2. Please note that with the latest NetBeans versions (7.0 and above) on-device debugging might not work correctly.

10.3.2 Integrating Cinterion WTK Manually

The integration of the Cinterion WTK into an appropriate NetBeans installation with added Mobility plugin can be done by the setup program automatically during first installation or afterwards in maintenance mode. Nevertheless it can be done although manually which is described in this chapter. To add the WTK open the NetBeans menu item "Tools" - "Java Platforms" and click the button "Add Platform…" in the following window.

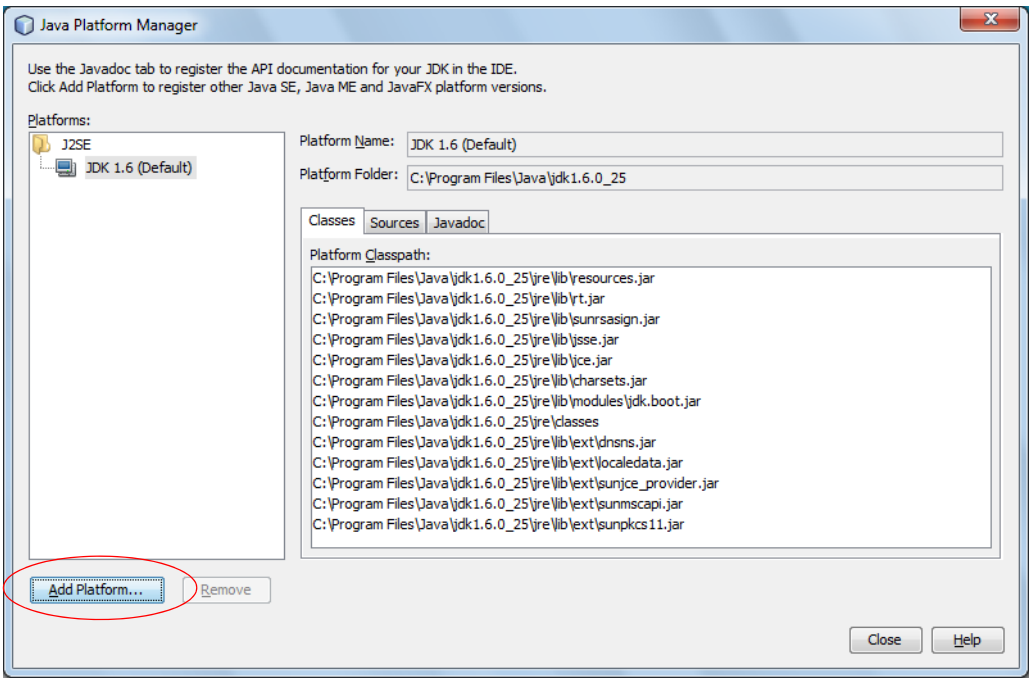


Figure 45: Integrating Cinterion WTK manually - Select platform

In the following dialog choose the platform type "Java ME MIDP Platform Emulator" and click the Next button.

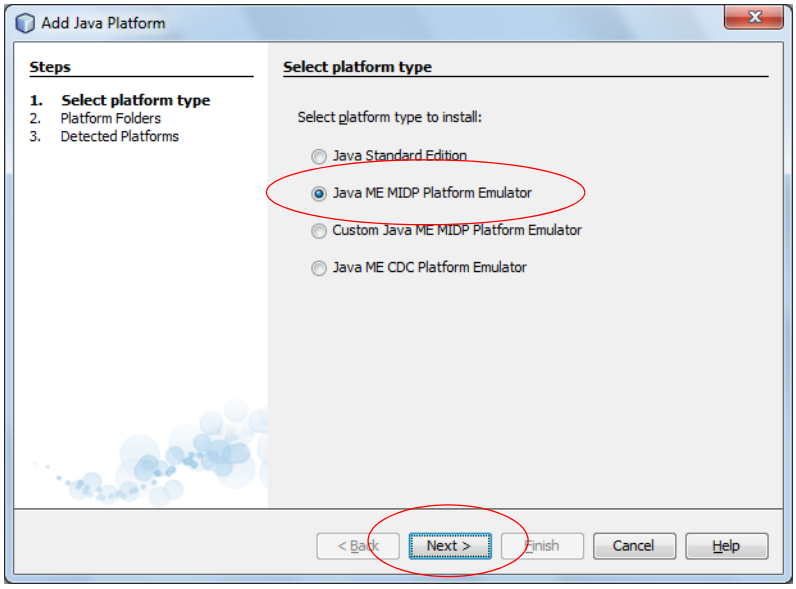


Figure 46: Integrating Cinterion WTK manually - Select type

In the file dialog browse to the root folder of the Cinterion WTK directory which is "Program Files\Cinterion\CMTK\ABC2\WTK". Now the Cinterion WTK is listed in the platform dialog.

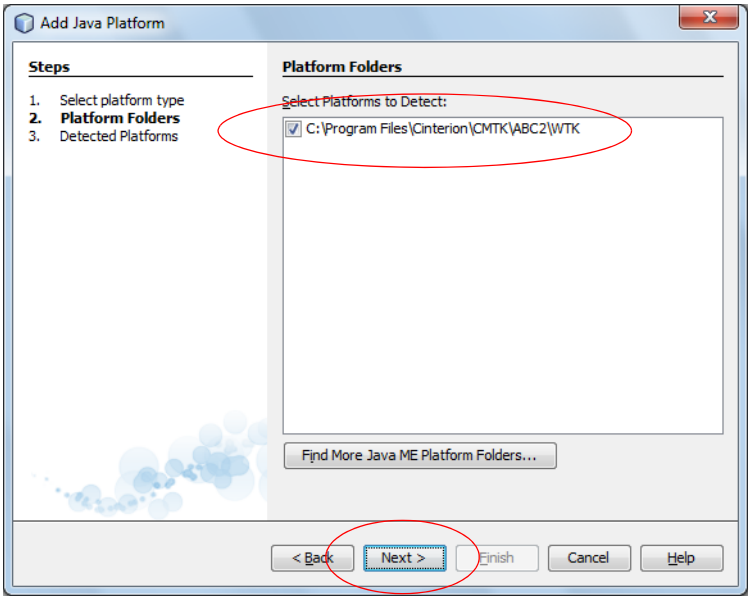


Figure 47: Integrating Cinterion WTK manually - Platform folder

After clicking the Next button the WTK interface is queried, the available emulator data displayed and the integration can be finished with the Finish button.

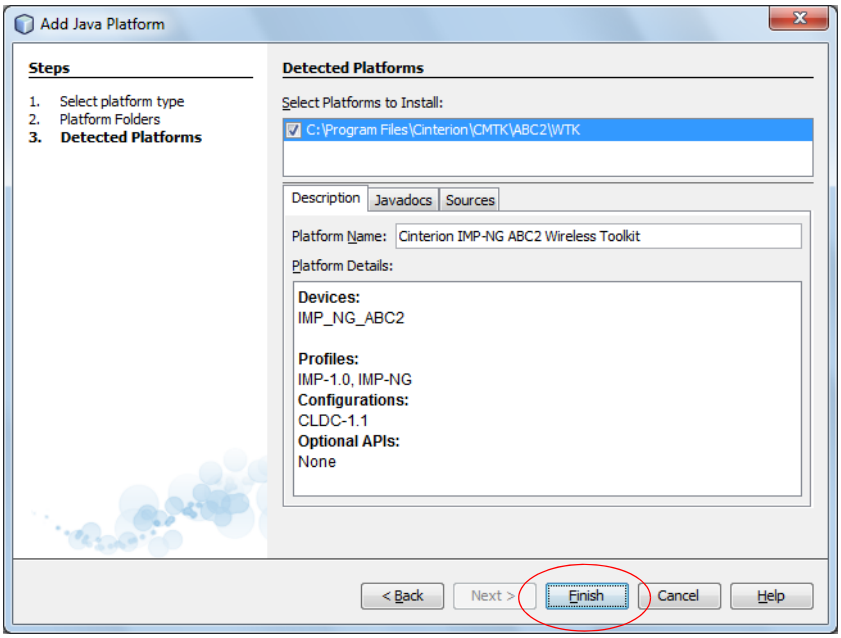


Figure 48: Integrating Cinterion WTK manually - Finish

10.3.3 Opening the Provided WTK Samples

The Cinterion WTK provides the existing samples in a NetBeans project format as well. They can simply be opened inside NetBeans via the menu item File-->"Open Project…" and navigating inside the open dialog to the sample location of either "Documents and Settings\All Users\Cinterion ABC2 WTK Examples\NetBeansSamples" under Windows XP or "Users\Public\Cinterion ABC2 WTK Examples\NetBeansSamples" under Windows Vista and above.

10.3.4 Creating a New MIDlet

To create a new MIDlet select the menu item File-->"New Project…". In the following window select the category "Java ME" and the project type "Mobile Application".

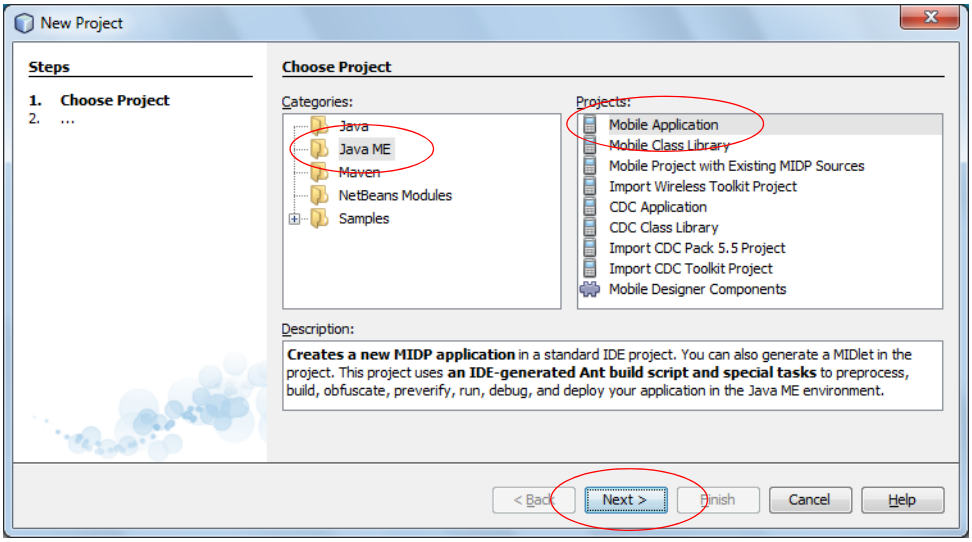


Figure 49: Creating a new MIDlet - Choose project

After clicking the Next button the project name and location can be entered. Please unselect the option "Create Hello MIDlet" because the provided MIDlet template does not fit the IMP profile very well.

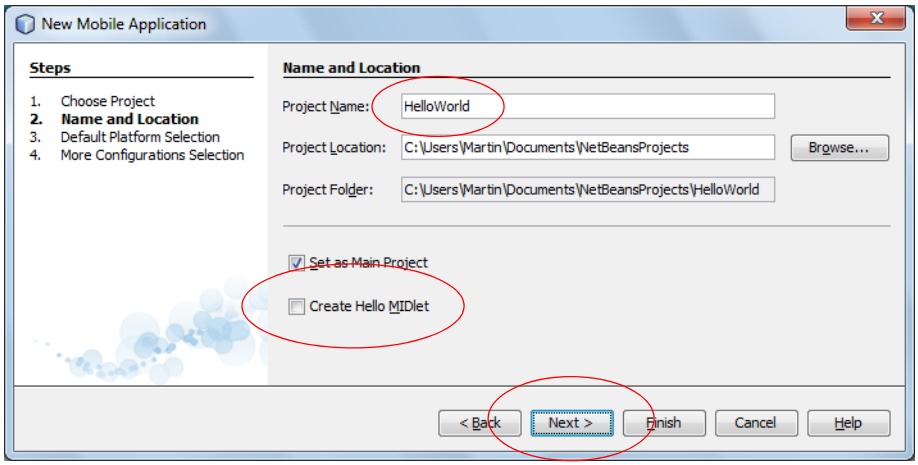


Figure 50: Creating a new MIDlet - Enter name and location

In the next dialog it must be ensured that the correct emulator platform "Cinterion IMP-NG ABC2 Wireless Toolkit", the correct device "IMP_NG_ABC2", the correct device configuration "CLDC-1.1" and the correct device profile "IMP-NG" are selected.

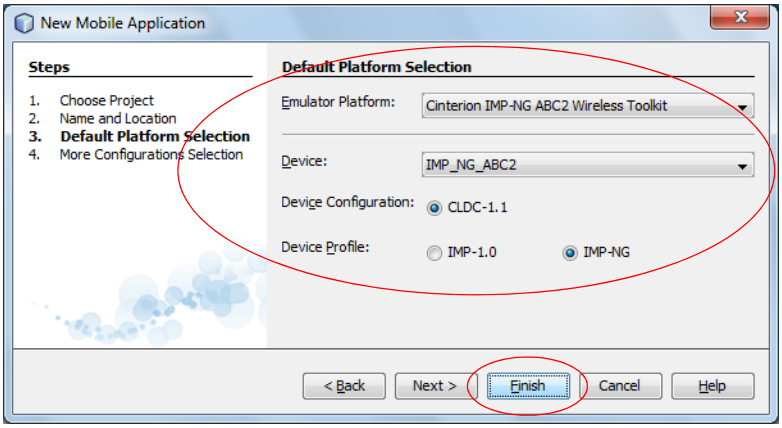


Figure 51: Creating a new MIDlet - Configure platform

After clicking the Finish button a newly created MIDlet project is available under the NetBeans project tree. To create the frame of a MIDlet program right click into the project tree and select New-->MIDlet… In the following wizard enter the name for the MIDlet and the MIDlet class and click the Finish button.

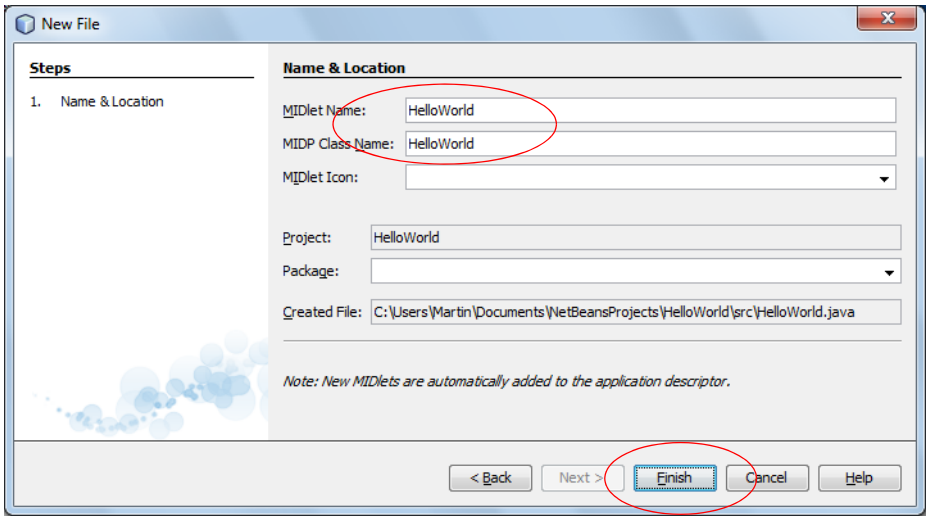


Figure 52: Creating a new MIDlet - Program name

Now, a corresponding Java file with the source of the basic MIDlet interface is created inside the source tree of the project. The only things that remains to be done is to add a call of "notifyDestroyed()" to the "destroyApp()" method and if required a class constructor.

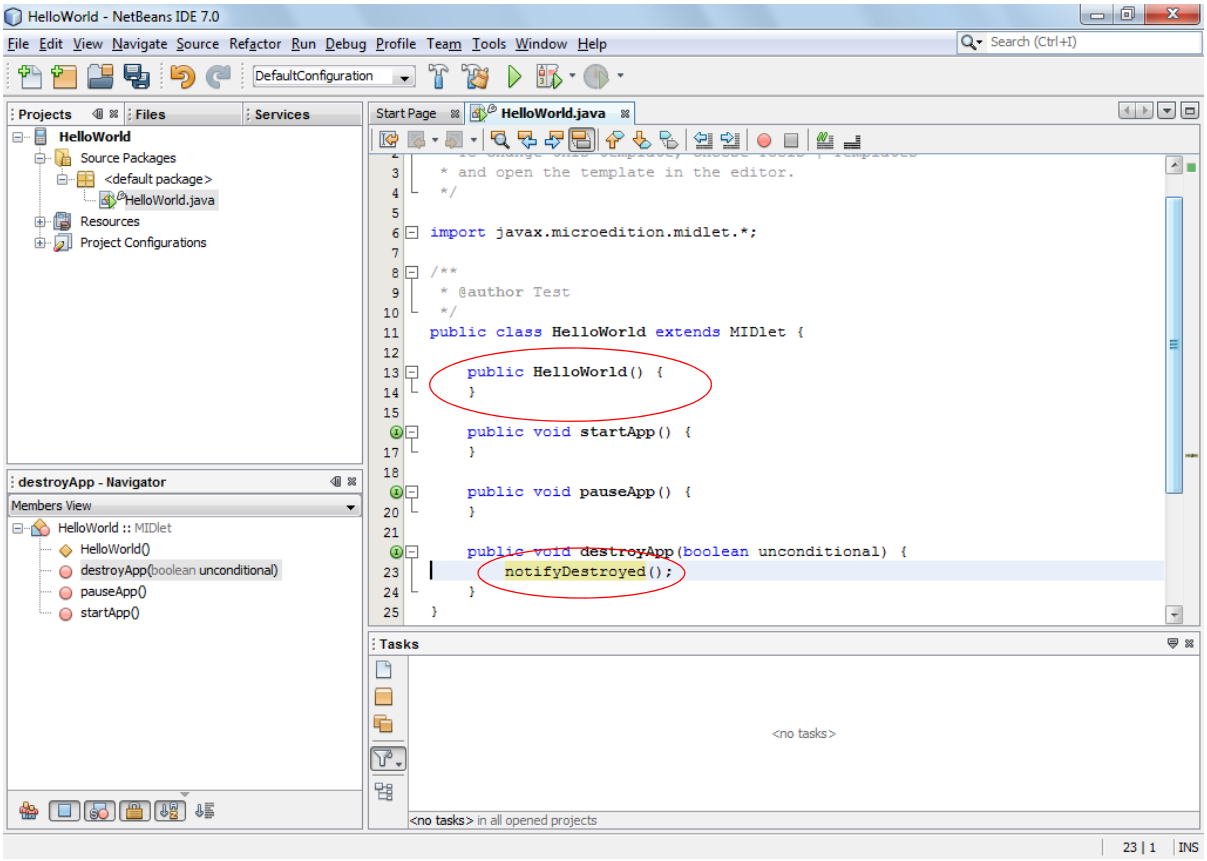


Figure 53: Creating a new MIDlet - HelloWorld.java

Finally, the actual functionality can be implemented and the project can be started and debugged inside the module as other NetBeans projects. Please note that you must allow NetBeans and the debug agent to pass the firewall and configure the debug IP connection as a home network to be able to execute and debug MIDlets inside the device.

10.4 Switching Java "System.out" to IDE Debug Window

The Java "System.out" can be redirected during a debugging session using a UDP socket connection as described in Section 5.9 and Section 5.9.3. Switching between Java "System.out" using serial port and Java "System.out" using UDP socket is done by a setting inside of the emulator configuration file "WM_Debug_config.ini" located in the WTK binary directory "Program Files\Cinterion\CMTK\ABC2\WTK\bin". Switching Java "System.out" to serial port is used as default setting.

This is the default "WM_Debug_config.ini" file as installed by the WTK setup:

```
#
# This ini file is used to configure emulator.exe
#

[General]
# Define the used module type
ModuleType=ABC2
ConnectionName=IP connection for remote debugging of ABC2

[AT command]
# These AT commands are used for initialising the module for debugging!
# Hints:
# ------
# The IP address range 10.x.x.x is not supported for configuration of debugging!
AT-Cmd1=S:ATE1
AT-Cmd2=R:OK
AT-Cmd3=S:at+cpin?
AT-Cmd4=R:+CPIN: READY
AT-Cmd5=R:OK
AT-Cmd6=S:at^scfg=userware/debuginterface,"192.168.0.2","192.168.0.1","0"
AT-CMd7=R:^SCFG: "Userware/DebugInterface","192.168.0.2","192.168.0.1","0"
AT-Cmd8=R:OK
AT-Cmd9=S:at^scfg=userware/mode,"debug","a:/","2000"
AT-Cmd10=R:^SCFG: "Userware/Mode","debug","a:/","2000"
AT-Cmd11=R:OK
AT-Cmd12=S:at^scfg=userware/stdout,UDP,
AT-Cmd13=R:^SCFG: "Userware/Stdout","UDP","
AT-Cmd14=R:OK

[Debug]
# used UDP port number range: 1024 ... 65535
# comment out the following line for switching off Java "System.out" displaying in the IDE window
#UDPport=12345
# settings for Debug Agent delay timer in ms
#DATimer=2000
```

The <UDPport> parameter of the "WM_Debug_config.ini" file is used for switching the Java "System.out" direction. To switch the Java "System.out" direction to the UDP socket and display it on a IDE window please remove "#" in front of the parameter. To switch the Java "System.out" direction to serial port output please add "#" (default setting).

Alternatively to modifying the "WM_Debug_config.ini" file it is possible to configure the UDP port via the emulator's command line parameter "-udpport:<port>". Additional emulator parameters for debugging can be configured through the Java IDEs. Please note that configuration via command line parameter overwrites a possible UDP port configuration of the ini file.

Please note that the handling of Java "System.out" is done asynchronously. It is possible that not all Java "System.out" data is written into the IDE window. To avoid this problem please set a breakpoint in front of the Java function call "notifyDestroyed()" inside your Java source. Also note that Java "System.out" using IDE window is only supported during debugging session.

11 Java Security

The Java Security Model follows the specification of MIDP 2.0 and is IMP-NG conforming. It integrates only a simple protection domain concept since protection domains are not needed for module use cases.

Java Security is divided into two main areas:
- Secure MIDlet data links (HTTPS, Secure Connection) (see Section 11.1)
- Execution of signed/unsigned MIDlets (see Section 11.2)

The interface of Java Security offers the following functionality.
- Insert/delete X.509 certificate (default is no certificate, see Section 11.2.1)
- Switch between trusted and untrusted mode for the execution of MIDlet (default is trusted after inserting the certificate, see Section 11.2.1)
- Enable/disable untrusted domain in trusted mode (default is disabled)
- Switch MES (default is ON see Section 11.3)
- Switch https certificate verification (default is OFF, see Section 11.1)

Restrictions:
- The module does not supply users independent date/time base. Therefore no examination of the validity of the expiration date/time of the certificate takes place.

11.1 Secure Data Transfer

This feature makes it possible for MIDlets to use safe data links to external communications partners. The specification IMP-NG defines two java classes with this characteristic - HTTPS-Connection and SecureConnection
.
The Cinterion implementation follows the recommendations in IMP-NG:

HTTPSConnection
- HTTP over TLS as documented in RFC 2818 and TLS Protocol Version 1.0 as specified in RFC 2246.

SecureConnection
- TLS Protocol Version 1.0 as specified in RFC 2246

Two modes exist for safe data links.

Mode 1:

- No examination of the server certificate takes place when setting up the connection. The authenticity of the server certificate is not verified. See Figure 54.

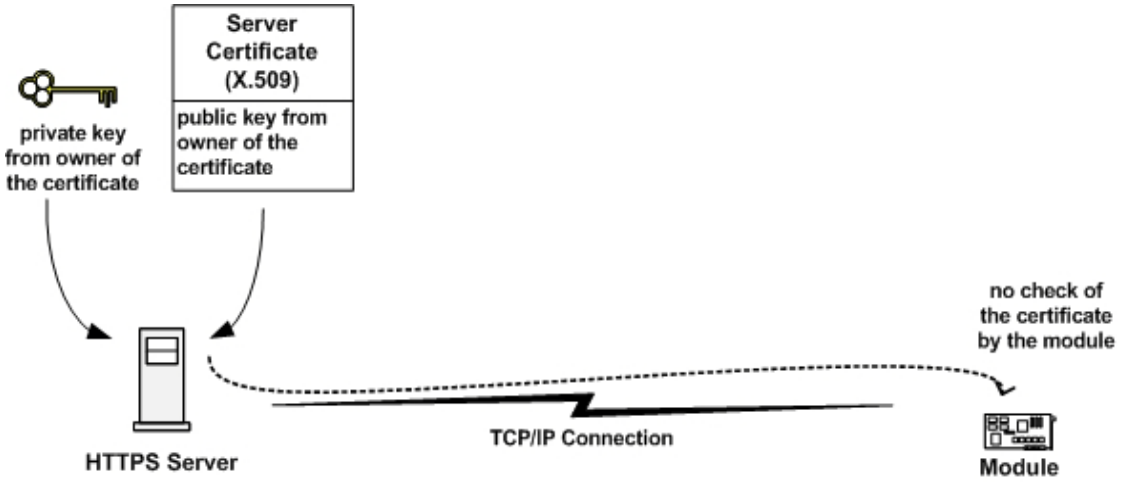


Figure 54: Mode 1 – Customer Root Certificate does not exist

Mode 2 (see Section 11.2.1, 1. Step*):*

- Customer Root Certificate is inside of the module.
- Command: Switch on Certificate Verification for HTTPS Connections was sent.
- The server certificate is examined when setting up a connection. Two configurations are valid. The server certificate is identical to the certificate in the module (both certificates are self signed root certificates) or the server certificate forms a chain with the certificate of the module. Thus the authenticity of the server certificate can be examined with the help of the certificate of the module. See Figure 55 and Figure 56.

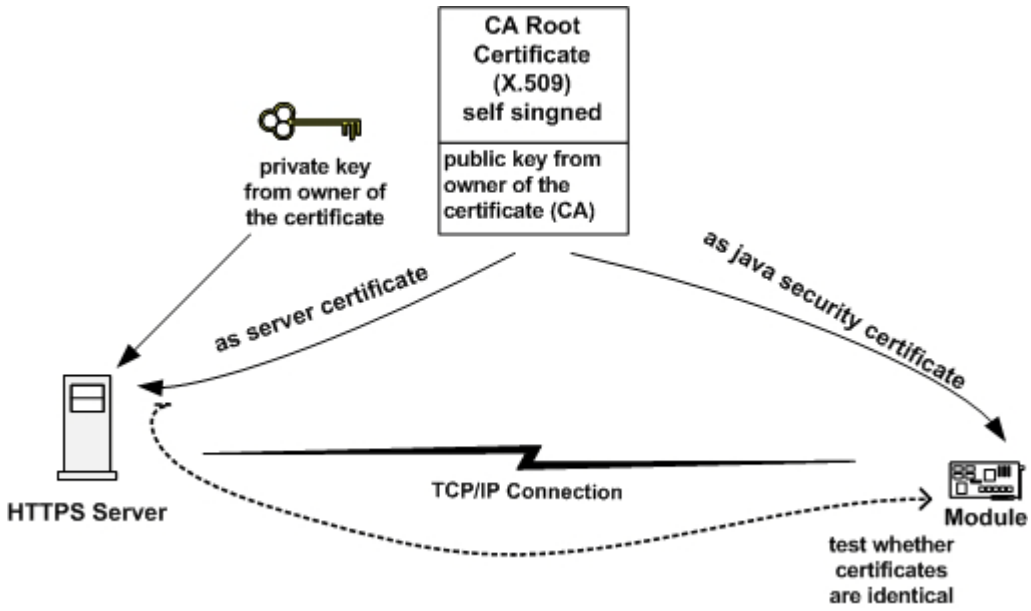


Figure 55: Mode 2 - Server Certificate and Certificate into module are identical

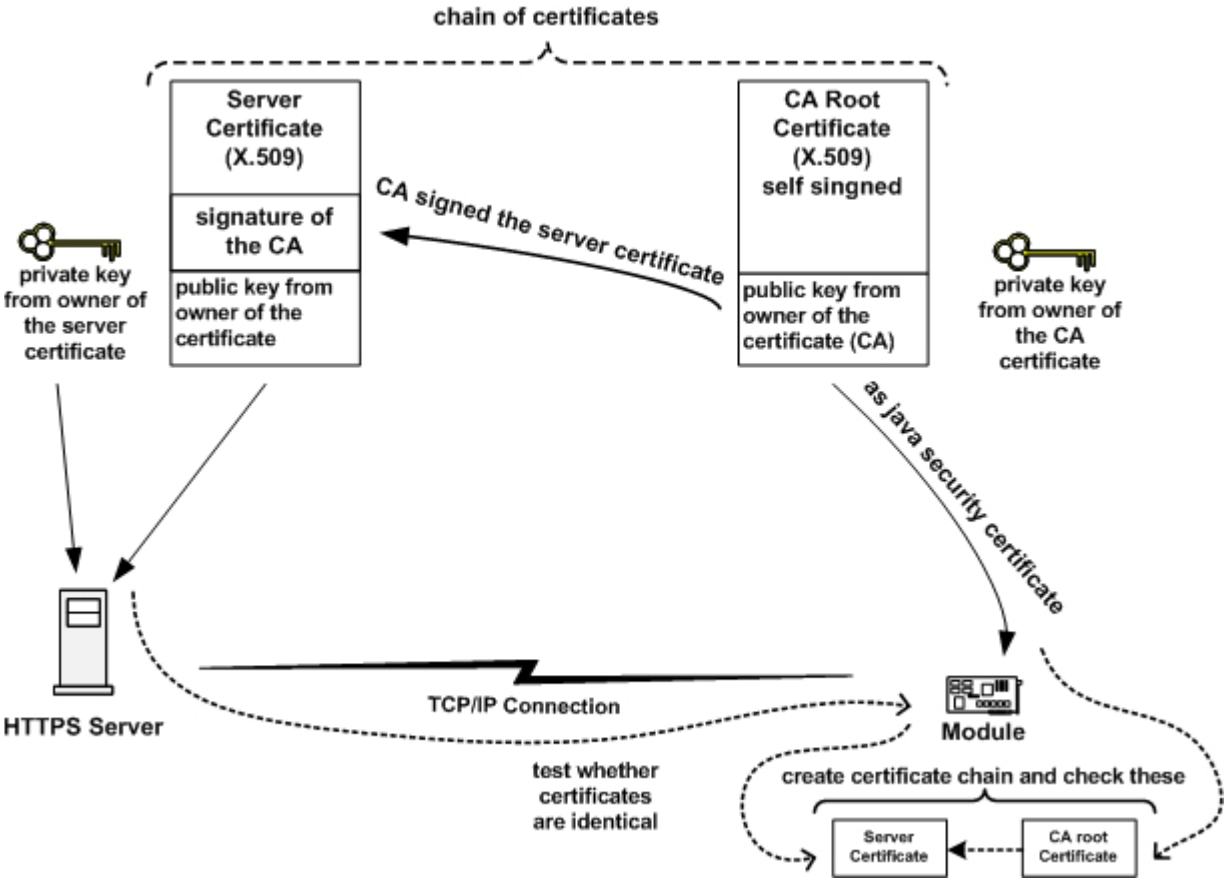


Figure 56: Mode 2 - Server Certificate and self signed root Certificate in module form a chain

11.1.1 Create a Secure Data Transfer Environment Step by Step

The following steps describe the configuration:
- The certificate exists within the module (*see* Section 11.2.1, Step 1).
- Certificate verification is activated for a data connection (HTTPS or SecureConnection).

The steps described below use the cygwin + openssl environment (for installation see http://www.cygwin.com/. The openssl documentation can be found here http://www.openssl.org/docs/apps/openssl.html)

A CA Root certificate is generated. This certificate can be placed on the HTTPS-Server. Another possibility is to use the private key of the certificate in order to sign thereby a server certificate. Both certificates form then a chain, which can be examined by the ME. A step-by-step description for the latter scenario can be found below.

1. Create CA and generate CA Root Certificate
   - We need certificates with sha1 signature. Java Security supports a sha1 signature of the certificate only.
     Add the parameter "-sha1" to the command "Making CA certificate ..." in the section of file CA.pl (cygwin location *cygwin**usr**ssl**misc"*)
   - Create a shell (use location *cygwin**usr**ssl**misc*)
   - Execute commands

     ```
     >perl CA.pl –newca
     ```

   - Convert file format from PEM to DER
     CA certificate cacert.pem

     ```
     >openssl x509 -in ./demoCA/cacert.pem -inform PEM
                    -out ./demoCA/cacert.der -outform DER
     ```

     CA private key file cakey.pem

     ```
     >openssl pkcs8 -in ./demoCA/private/cakey.pem
                     -inform PEM -out ./demoCA/private/cakey.der
                     -outform DER -nocrypt -topk8
     ```

2. Create server certificate and java keystore
   - Execute command

     ```
     >keytool -genkey -alias server
              -keypass keypass -keystore customer.ks -storepass keystorepass
               -sigalg SHA1withRSA -keyalg RSA
     ```

   The field "name" of the certificate is the domain name or the IP address of the server.

3. Create certificate request for server certificate
   - Execute command

     ```
     >keytool -certreq -alias server -file server.csr
              -keypass keypass -keystore customer.ks
              -storepass keystorepass
     ```

4. Sign certificate request by CA
   - Execute command

     ```
     >openssl ca -in server.csr -out server.pem
     ```

   - Convert file format from PEM to DER

     ```
     >openssl x509 -in server.pem -inform PEM
                    -out server.der -outform DER
     ```

5. Import CA root certificate and CA private key into java keystore
   - Use the CA Root Certificate for the creation of Java Security Command. See Section 11.5.3.
   - Execute command

```
>java -jar setprivatekey.jar -alias dummyca
        -storepass keystorepass -keystore customer.ks
        -keypass cakeypass
        -keyfile ./democa/private/cakey.der -certfile ./democa/cacert.der
```

6. Export private key from server certificate
   - The private key is needed for the (HTTPS or Secure Connection)server configuration.
   - Execute command

```
java -jar getprivatekey.jar -alias server
      -keystore customer.ks -storepass keystorepass
      -keypass keypass -keyfile server_privkey.der
```

   - Convert format

```
>openssl pkcs8 -in server_privkey.der
                    -inform DER -out server_privkey.pem
                    -outform PEM -nocrypt
```

7. Send CA Root Certificate to the module
   - Create Java Security Command "**SetRootCert"**. See Section 11.5.3.
   - Send this command with AT^SJSEC to the module. See Section 11.4.3.

8. Send Java Security Command "Switch on Certificate Verification for HTTPS Connections" to the module. See Section 11.5.3.

Result:
- You have a keystore for the configuration of the Java Security of the module.
- You have a signed server certificate (files "server.pem" or "server.der").
- You have a private key file for your server configuration (files "server_privkey.pem" or "server_privkey.der.
- The module contains the CA Root Certificat.
- HTTPS certificate verification is activate.

Consult the documentation of your web server on how to set.up the certificate on it.

11.2 Execution Control

The Java environment of the ME supports two modes.

Unsecured mode:
- The device starts all Java applications (MIDlets).

Secured mode:
- A condition for the secured mode of the device is the existence of a certificate inside of the module.
- The customer can activate the secured mode of the device. To do so, the customer sends a root certificate (x.509 certificate) and the command Switch ON Security Mode to the device (over an AT Interface). The device changes from unsecured mode to the secured mode. From this time the module will only start Java applications with a valid signature. In addition, the device will only accept special commands from the customer if they are marked with a signature. The device examines each command with the public key of the customer root certificate.
- The secured mode supports a simple protection domain concept, providing a domain for unsigned MIDlets. If this domain (domain for untrusted MIDlets) is active, then an unsigned MIDlet is assigned to this domain and has only limited access to the Java-API. The untrusted domain is activated by use of Java Security Command Switch ON Untrusted Domain (see Section 11.5.3).

  untrusted domain:
  - HTTP-Connection is permitted
  - TCP/IP-Socket Connection is permitted

Standard behavior of the module:
Cinterion supplies modules with unsecured mode as the default configuration.
Insert the certificate:
- The module changes into the mode "trusted" for MIDlet execution. "Untrusted Domain" is OFF.
- HTTPS certificate verification is OFF.
- MES is ON.

Remove the certificate:
- The module changes into the mode "untrusted" for MIDlet execution.
- HTTPS certificate verification is OFF
- MES is ON.

11.2.1 Change to Secured Mode Concept

Create and insert a customer root certificate:
A condition for the change to the secured mode is the existence of a customer root certificate inside of the module.

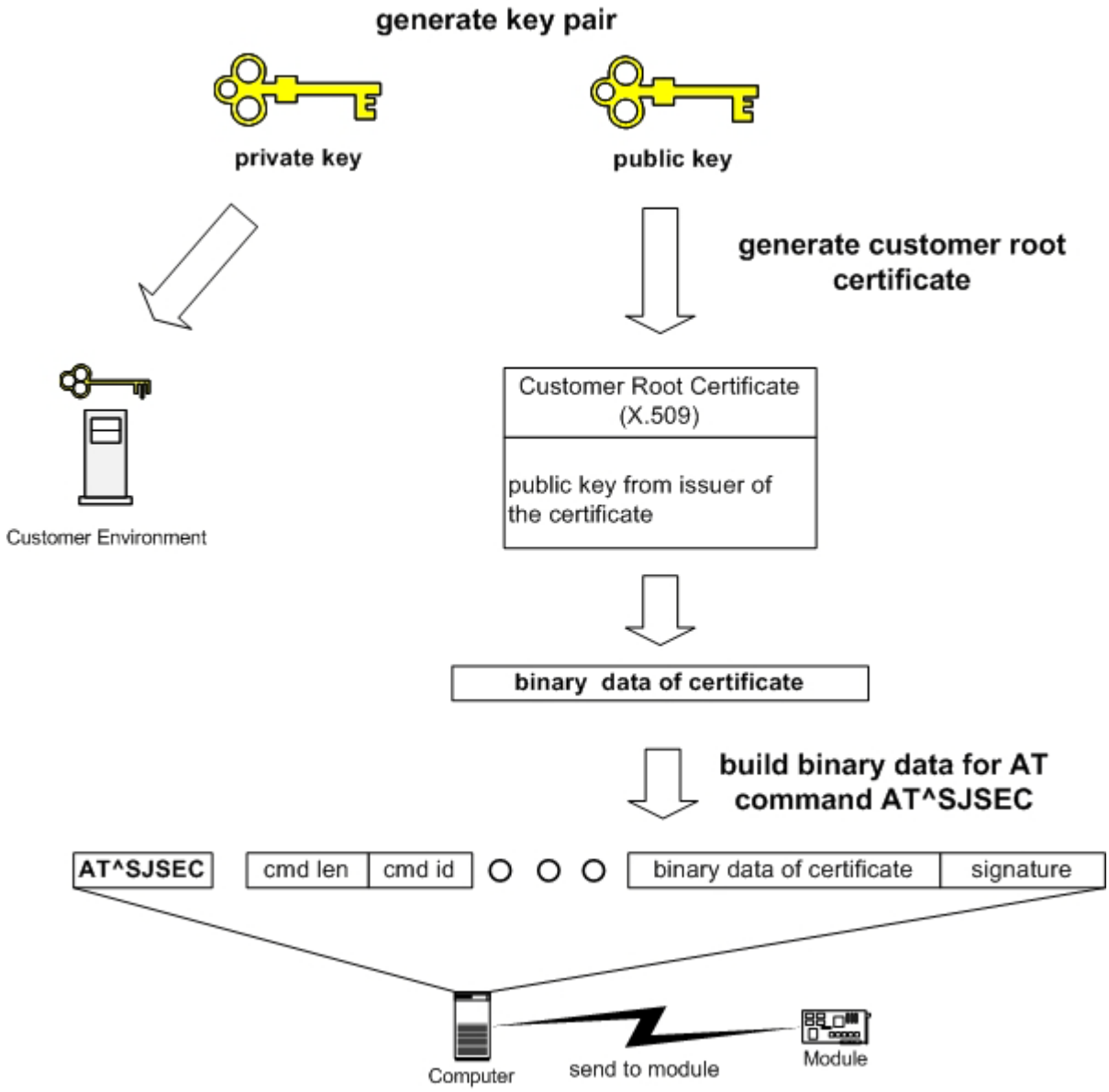


Figure 57: Insert Customer Root Certificate

After this action the module is in the following conditions:

- The module changes into the mode "trusted" for MIDlet execution. "Untrusted Domain" is OFF.
- HTTPS certificate verification is OFF.
- MES is ON.

11.2.2 Concept for the Signing the Java MIDlet

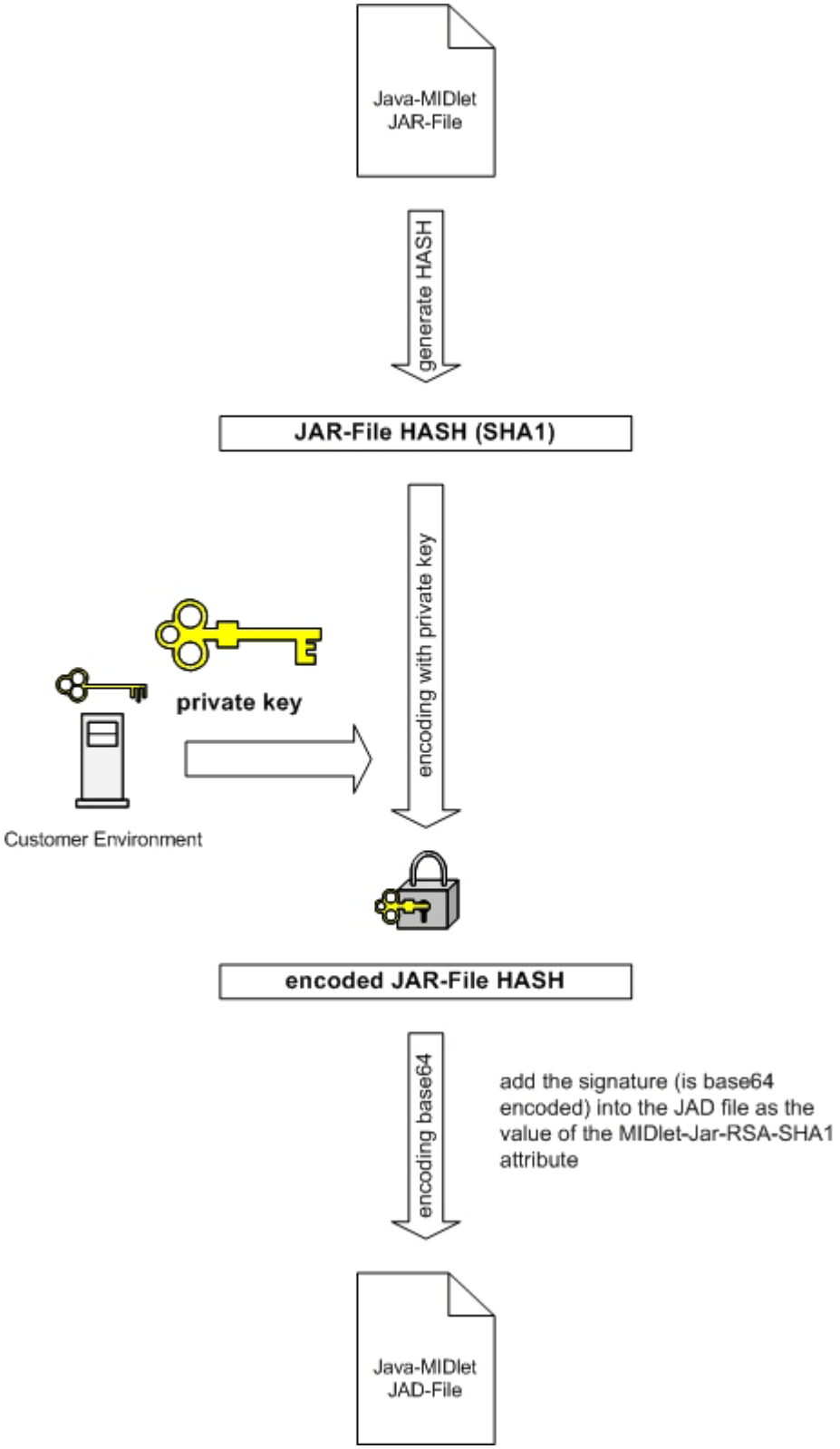


Figure 58: Prepare MIDlet for Secured Mode

11.3 Application and Data Protection

In addition to the Java secured mode it is possible to prevent the activation of the Module Exchange Suite. When Module Exchange Suite access is deactivated with *AT^SJSEC*, it is no longer possible to access to the Flash file system on the module. A condition for the deactivation of the access to the Flash file system is the existence of a customer root certificate inside of the module (see Section 11.2.1).

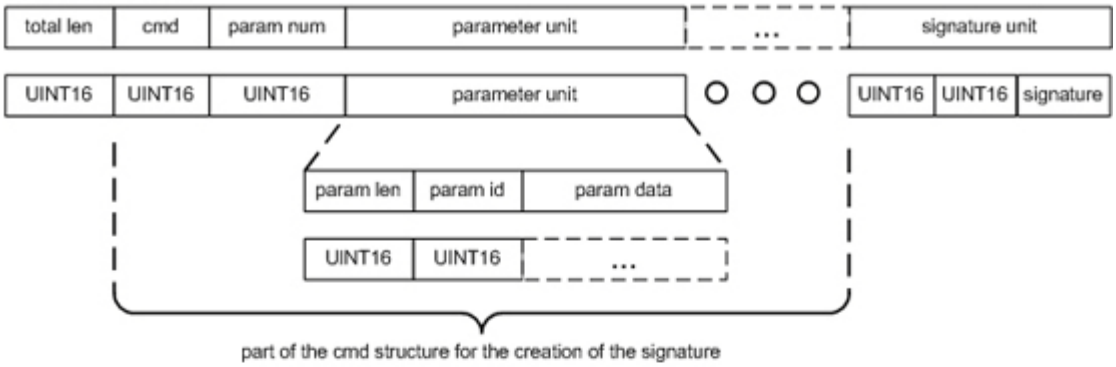
The default state of MES is ON.

11.4 Structure and Description of the Java Security Commands

Special commands are used in the Java security environment. These commands are transferred to the module with the help of the special AT command *AT^SJSEC*. This command makes it possible to send binary data to the module. After *AT^SJSEC* is issued, the module changes into a block transfer mode. Now binary data in a fixed format can be sent. These binary data contain the actual Java security commands.

11.4.1 Structure of the Java Security Commands

General structure



total len = all bytes of the command structure (including size of "total len")

param len = all bytes of the parameter structure (including size of "param len")

List of parameters

param id	param len	param data	description
0x0001	D	certificate data	content of *.der file
0x0002	0x0005	0x00 or 0x01	on/off switch, 0x00 = off, 0x01 = on
0x0003	0x0014	IMEI	numeric numbers in ASCII format (zero terminated string)
0x0004	D	signature data	SHA-1signature the of command, base64 coded (zero terminated string)

D - depend from the length of parameter

List of commands

cmd id	description
0x0001	Set Customer Root Certificate
0x0002	Del Customer Root Certificate
0x0003	Switch on/off Certificate Verification for HTTPS Connections
0x0004	Switch on/off OBEX Functionality
0x0005	Switch on/off Security Mode of the module
0x0006	Switch on/off Untrusted Domain inside of the Security Mode

Set Customer Root Certificate

total len	0x0001	0x0003	param unit certificate	param unit IMEI	param unit signature

Del Customer Root Certificate

total len	0x0002	0x0002	param unit IMEI	param unit signature

Switch on/off Certificate Verification for HTTPS Connections

total len	0x0003	0x0003	param unit switch	param unit IMEI	param unit signature

Switch on/off OBEX Functionality

total len	0x0004	0x0003	param unit switch	param unit IMEI	param unit signature

Switch on/off Security Mode

total len	0x0005	0x0003	param unit switch	param unit IMEI	param unit signature

Switch on/off Untrusted Domain

total len	0x0006	0x0003	param unit switch	param unit IMEI	param unit signature

11.4.2 Build Java Security Command

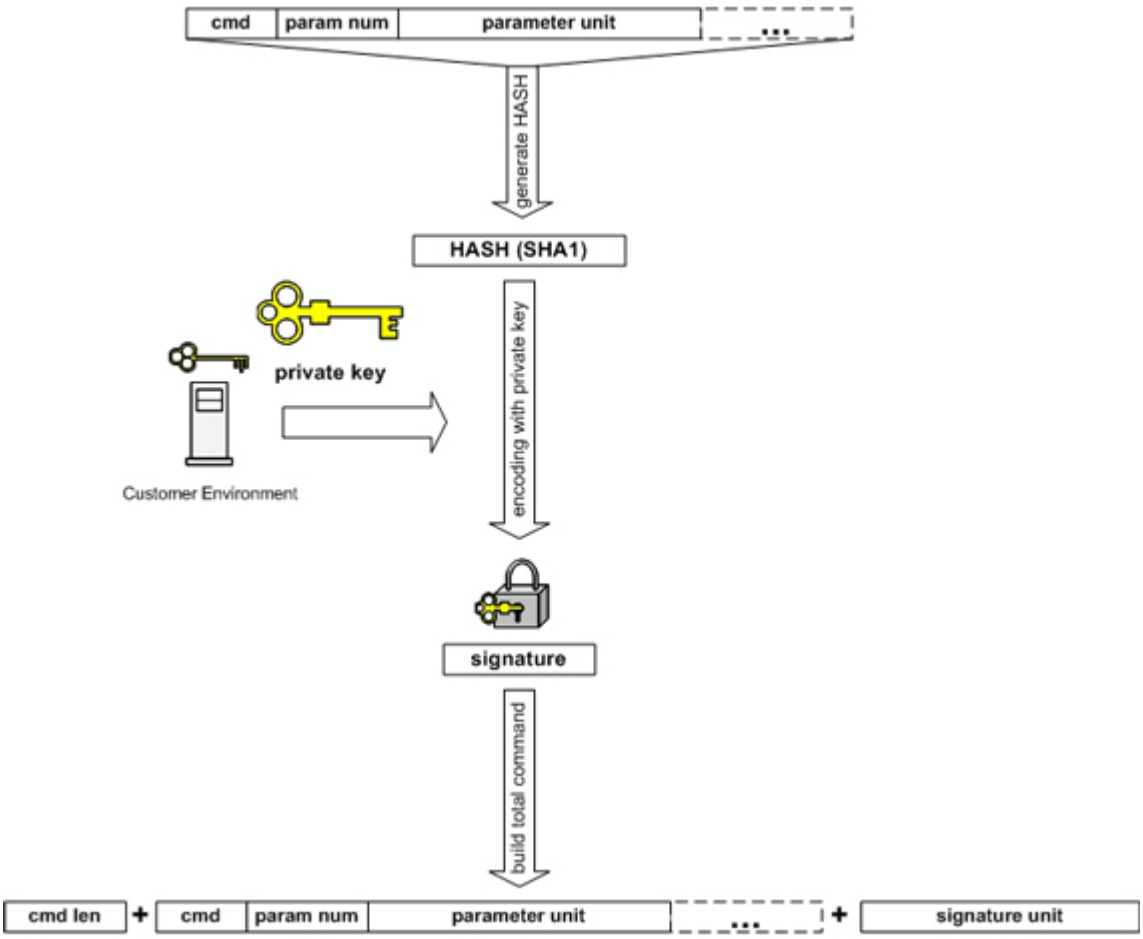


Figure 59: Build Java Security Command

11.4.3 Send Java Security Command to the Module

Use a terminal program. Enter:
AT^SJSEC
Wait for the answer:
CONNECT
JSEC READY: SEND COMMAND ...

Now you can send the binary data of the command (for example: from a file with the binary data of the command). To abort send: 0x03 0x00 0x00 (three bytes) in one packet.

The module's answer depends on the result of the operation.
The read command, AT^SJSEC?, can be used to request the current Java security status. The responses are listed below:

Read command AT^SJSEC?	Response: ^SJSEC:<state>,<HTTPS state>,<OBEX state>,<untrusted domain> [<certificate content>] <state> Java security mode 0 Unsecured mode, i.e. security mode not active (default) 1 Secured mode, i.e. security mode active <HTTPS state> 0 The HTTPS connection or Secure Connection is possible if the server certificate (or certificate chain) is valid (default) 1 The HTTPS Connection or Secure Connection is possible only if the server certificate is signed by the customer (owner of root certificate in device) <OBEX state> 0 Start of Module Exchange Suite is not permitted 1 Start of Module Exchange Suite is permitted (default) <untrusted domain> 0 Untrusted domain does not exist, MIDlets must be signed (default) 1 Untrusted domain does exist, unsigned MIDlets have limited rights of access to the Java-API Certificate Information: (default no certificate) Issuer: SerialNumber: Subject: Signature algorithm: Thumbprint algoritm: Thumbprint:

11.5 Create a Java Security Environment Step by Step

11.5.1 Create Key Store

The key store contains the key pairs for signing data. For producing the key store with keys the tool "keytool.exe" can be used. The program is in the Java SDK. (for a description see http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html).
Example:

```
keytool -genkey -alias keyname -keypass keypassword -keystore customer.ks
        -storepass keystorepassword -sigalg SHA1withRSA -keyalg RSA
```

If you want to re-use the key store created in Section 11.1.1 you have to use the alias and passwords used there.

11.5.2 Export X.509 Root Certificate

For exporting the x.509 root certificate use "keytool.exe". The program is in the Java SDK. (for description see http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html)

```
keytool -export -v -keystore customer.ks -storepass keystorepassword
        -alias keyname > certificate.der
```

11.5.3 Create Java Security Commands

For producing the java security commands the tool "jseccmd.jar" can be used. This program is in the folder "wkt\bin".

Command for inserting the Customer Root Certificate

- In consequence of the command
  Java security mode: ON (untrusted domain OFF)
  HTTPS certificate verification: OFF
  MES state: ON

```
java -jar jseccmd.jar -cmd SetRootCert
                      -certfile certificate.der
                      -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > SetRootCert.bin
```

Command for removing the Customer Root Certificate

- In consequence of the command all switches are reset
  Java security mode: OFF (untrusted domain OFF)
  HTTPS certificate verification: OFF
  MES state: ON

```
java -jar jseccmd.jar  -cmd DelRootCert
                       -imei 012345678901234 -alias keyname
                       -storepass keystorepassword -keypass keypassword
                       -keystore customer.ks > DelRootCert.bin
```

Command: switch on Java Security Mode

```
java -jar jseccmd.jar -cmd TrustedModeOn
                      -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > TrustedModeOn.bin
```

Command: switch off Java Security Mode

```
java -jar jseccmd.jar -cmd TrustedModeOff
                       -imei 012345678901234 -alias keyname
                       -storepass keystorepassword -keypass keypassword
                       -keystore customer.ks > TrustedModeOff.bin
```

Command: switch on Untrusted Domain (it is possible only inside of the Java Security Mode)

```
java -jar jseccmd.jar -cmd UntrustedDomainOn
                      -imei 012345678901234 -alias keyname
                        -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > UntrustedDomainOn.bin
```

Command: switch off Untrusted Domain

```
java -jar jseccmd.jar -cmd UntrustedDomainOff
                      -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > UntrustedDomainOff.bin
```

Command: switch on Certificate Verification for HTTPS Connections

```
java -jar jseccmd.jar -cmd HttpsVerifyOn
                      -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > HttpsVerifyOn.bin
```

Command: switch off Certificate Verification for HTTPS Connections

```
java -jar jseccmd.jar -cmd HttpsVerifyOff
                      -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > HttpsVerifyOff.bin
```

Command: switch on module exchange functionality

```
java -jar jseccmd.jar -cmd ObexActivationOn
                        -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > ObexActivationOn.bin
```

Command: switch off module exchange functionality

```
java -jar jseccmd.jar -cmd ObexActivationOff
                        -imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > ObexActivationOff.bin
```

11.5.4 Sign a MIDlet

Use the tool "jadtool.jar" to sign a Java MIDlet. This program is in the folder "wkt\bin".

```
java -jar jadtool.jar -addjarsig -jarfile helloworld.jar
                      -inputjad helloworld.jad
                      -outputjad helloworld.jad
                      -alias keyname -storepass keystorepassword
                      -keypass keypassword -keystore customer.ks
                      -encoding UTF-8
```

11.6 Attention

The central element of Java Security is the **private key**. If Java Security is activated and you lose the private key, then the **module is destroyed**. You have no chance of deactivating Java Security, downloading of a new Midlet or starting any other operation concerning Java Security. To prevent problems you are strongly advised to **secure the private key**.

12 Java Tutorial

This small tutorial includes explanations on how to use the AT Command API and suggestions for programming MIDlets. The developer should read about MIDlets, Threads and AT commands as a complement to this tutorial.

12.1 Using the AT Command API

Perhaps the most important API for the developer is the AT command API. This is the API that lets the developer issue commands to control the module. This API consists of the *ATCommand* class and the *ATCommandListener* and *ATCommandResponseListener* interfaces. Their javadocs can be found in …*wtk**doc**html**index.html,* [3].

12.1.1 Class ATCommand

The *ATCommand* class supports the execution of AT commands in much the same way as they would be executed over a serial interface. It provides a simple way to send strings directly to the device's AT parsers.

12.1.1.1 Instantiation with or without CSD Support

There can be only exactly as many *ATCommand* instances as there are parsers on the device. If there are no more parsers available, the *ATCommand* constructor will throw *ATCommandFailedException*. All AT parser instances support CSD. However from a Java application point of view it may make sense to have one dedicated instance for CSD call handling. Therefore, and also for historical reasons, only one parser with CSD support may be requested through the constructor. If more then one parser with CSD support is requested, the constructor will throw *ATCommandFailedException.*

```
try {
ATCommand atc = new ATCommand(false);
  /* An instance of ATCommand is created. CSD is not explicitly
   * requested. */
}catch (ATCommandFailedException e) {
System.out.println(e);
}
```

The csdSupported() method returns the CSD capability of the connected instance of the device's AT parser. The method checks as well, if the current mode of the module supports CSD. Please notice that this check has not been done when opening the *ATCommand* instance.

```
boolean csd_support = atc.csdSupported();
```

release() releases the resources held by the instance of the ATCommand class. After calling this function the class instance cannot be used any more but the resources are free to be used by a new instance

12.1.1.2 Sending an AT Command to the Device, the send() Method

An AT command is sent to the device by using the send() method. The AT command is sent as a string which must include the finalizing line feed "\r" or the corresponding line end character.

```
String response = atc.send("at+cpin?\r");
/* method returns when the module returns a response */
System.out.println(response);

#Possible response printed to System.out:
+CPIN: READY OK
```

This send function is a blocking call, which means that the calling thread will be interrupted until the module returns a response. The function returns the response, the result code of the AT command, as a string.

Occasionally it may be infeasible to wait for an AT command that requires some time to be processed, such as AT+COPS. There is a second, non-blocking, send function which takes a second parameter in addition to the AT command. This second parameter is a callback instance, *ATCommandResponseListener*. Any response to the AT command is delivered to the callback instance when it becomes available. The method itself returns immediately. The *ATCommandResponseListener* and the non-blocking send method are described in Section 12.1.2.

Note: Using the send methods with strings with incorrect AT command syntax will cause errors.

12.1.1.3 Data Connections

If a data connection is created with the *ATCommand* class, for instance with *ATD* an input stream is opened to receive the data from the connection. Similarly, an output stream can be opened to send data on the connection.

```
/* Please note that this example would not work unless the module had
 * been initialized and logged into a network. */

System.out.println("Dialing: ATD" + CALLED_NO);
response = atc.send("ATD" + CALLED_NO + "\r");
System.out.println("received: " + response);

if (response.indexOf("CONNECT") >= 0) {
  try {
    // We have a data connection, now we do some streaming...
    // IOException will be thrown if any of the Stream methods fail
    OutputStream dataOut = ATCmd.getDataOutputStream();
    InputStream dataIn = ATCmd.getDataInputStream();

    // out streaming...
    dataOut.write(new String("\n\rHello world\n\r").getBytes());
    dataOut.write(new String("\n\rThis data was sent by a Java " +
                        "MIDlet!\n\r").getBytes());
    dataOut.write(new String("Press 'Q' to close the " +
                        "connection\n\r").getBytes());



    // ...and in streaming
    System.out.println("Waiting for incoming data, currently " +
                       dataIn.available() + " bytes in buffer.");
    rcv = 0;
    while(((char)rcv != 'q') && ((char)rcv != 'Q') && (rcv != -1)){
      rcv = dataIn.read();
      if (rcv >= 0) {
        System.out.print((char)rcv);
      }
    }       /* The example continues after the next block of text */
```

These streams behave slightly differently than regular data streams. The streams are not closed by using the close() method. A stream remains open until the *release()* method is called. A module can be switched from the data mode to the AT command mode by calling the *breakConnection()* method.

```
/* continue example */
   if (rcv != -1) {
     // Now break the data connection
     System.out.println("\n\n\rBreaking connection");
     try {
       strRcv = ATCmd.breakConnection();
     } catch(Exception e) {
       System.out.println(e);
     }
     System.out.println("received: " + strRcv);
   } else {
     // Received EOF, somebody else broke the connection
     System.out.println("\n\n\rSomebody else switched to " +
                        "command mode!");
   }
   System.out.println("Hanging up");
   strRcv = ATCmd.send("ATH\r");
   System.out.println("received: " + strRcv);
  } catch(IOException e) {
   System.out.println(e);
  }
} else {
  System.out.println("No data connection established,");
}
```

An IOException is thrown if any function of the I/O streams are called when the module is in AT command mode, except for read functions. The read function case is explained in more detail in Section 12.1.3. The ATCommand class does not report the result codes returned after data connection release.

Data Connections are not only used for data transfer over the air but also to access external hardware. Here is a list of at commands which open a data connection (see [1] for details):

- ATD for data calls
- AT^SSPI, for access to I2C/SPI
- Several AT commands for Internet services

For data connection signaling see also Section 12.1.3.

12.1.1.4 Synchronization

For performance reasons no synchronization is done in the ATCommand class. If an instance of this class has to be accessed from different threads ensure that the *send()* functions, the *release()* function, the *cancelCommand()* function and the *breakConnection()* function are synchronized in the user implementation.

12.1.2 ATCommandResponseListener Interface

The *ATCommandResponseListener* interface defines the capabilities for receiving the response to an AT command sent to one of the module's AT parsers. When the user wants to use the non blocking version of the *ATCommand.send* function of an implementation class of the *ATCommandResponseListener* interface must be created first. The single method of this class, *ATResponse()*, must contain the processing code for the possible response to the AT command sent.

```
class MyListener implements ATCommandResponseListener {

  String listen_for;

  public MyListener(String awaited_response) {
    listen_for = awaited_response;
  }

  void ATResponse(String Response) {
    if (Response.indexOf(listen_for) >= 0) {
      System.out.println("received: " + strRcv);
    }
  }
}
```

12.1.2.1 Non-Blocking ATCommand.send() Method

After creating an instance of the *ATCommandResponseListener* class, the class instance can be passed as the second parameter of the non-blocking ATCommand.send() method. After the AT command has been passed to the AT parser, the function returns immediately and the response to the AT command is passed to this callback class later when it becomes available Somewhere in the application:

```
MyListener connect_list = new MyListener("CONNECT");
atc.send("ATD" + CALLED_NO + "\r", connect_list);

/* Application continues while the AT command is processed*/
/* When the module delivers its response to the AT command the callback
 * method ATResponse is called. If the response is "CONNECT", we will see
 * the printed message from ATResponse in MyListener. */
```

A running AT command sent with the non-blocking send function can be cancelled with *ATCommand.cancelCommand()***.** Any possible responses to the cancellation are sent to the waiting callback instance.

Note: Using the send methods with incorrect AT command syntax in the strings will cause errors.

12.1.3 ATCommandListener Interface

The *ATCommandListener* interface implements callback functions for:
- URCs
- Changes of the serial interface signals RING, DCD and DSR
- Opening and closing of data connections

The user must create an implementation class for *ATCommandListener* to receive AT events. The *ATEvent* method of this class must contain the processing code for the different AT-Events (URCs). The *RINGChanged, DCDChanged, DSRChanged* and *CONNChanged* methods should contain the processing code for possible state changes. This code shall leave the listener context as soon as possible (i.e. running a new thread). While the callback method does not return it cannot be called again if changes occur.

The *CONNChanged* method indicates the start and the end of data connections. During a data connection it is possible to transfer data with the I/O stream methods (see Section 12.1.1.3). Some data services (i.e. FTP) transfer the data so quickly, that the *CONNChanged* start and close events are received even parallel to the response of the AT command which originated the data connection. The user's application must realize, that the data connection had taken place and read the data with the I/O stream read methods afterwards.

12.1.3.1 ATEvents

An ATEvent or a URC is a report message sent from the module to the application. An unsolicited result code is either delivered automatically when an event occurs or as a result of a query the module previously received. However, a URC is not issued as a *direct* response to an executed AT command. Some URCs must be activated with an AT command.

Typical URCs may be information about incoming calls, a received SM, temperature changes, the status of the battery, etc. A summary of URCs is listed in [1].

12.1.3.2 Implementation

```
class ATListenerA implements ATCommandListener {

public void ATEvent(String Event) {
  if (Event.indexOf("+CALA: Reminder 1") >= 0) {
    /* take desired action after receiving the reminder */
  } else if (Event.indexOf("+CALA: Reminder 2") >= 0) {
    /* take desired action after receiving the reminder */
  } else if (Event.indexOf("+CALA: Reminder 3") >= 0) {
    /* take desired action after receiving the reminder */
  }
/* No action taken for these events */
  public void RINGChanged(boolean SignalState) {}
  public void DCDChanged(boolean SignalState) {}
  public void DSRChanged(boolean SignalState) {}
}

class ATListenerB implements ATCommandListener {

  public void ATEvent(String Event) {
    if (Event.indexOf("+SCKS: 0") >= 0) {
      System.out.println("SIM Card is not inserted.");
      /* perform other actions */
    } else if (Event.indexOf("+SCKS: 1") >= 0) {
      System.out.println("SIM Card is inserted.");
     /* perform other actions */
    }
  }
  public void RINGChanged(boolean SignalState) {
    /* take some action when the RING signal changes if you want to */
  }

  public void DCDChanged(boolean SignalState) {
    /* take some action when the DCD signal changes if you want to */
  }

  public void DSRChanged(boolean SignalState {}
    /* take some action when the DSR signal changes if you want to */
  }

  public void CONNChanged(boolean SignalState {}
    /* take some action when the state of a connection changes if you want
    to */
  }
}
```

12.1.3.3 Registering a Listener with an ATCommand Instance

After creating an instance of the *ATCommandListener* class, it must be passed as a parameter to the *ATCommand.addListener()* method. The callback methods of the instance will be called by the runtime system each time the corresponding events (URCs or signal state changes) occur on the corresponding device AT parser.

```
/* we have two ATCommands instances, atc1 and atc2 */
ATListenerA reminder_listener = new ATListenerA();
ATListenerB card_listener = new ATListenerB();
atc1.addListener(reminder_listener);
atc2.addListener(card_listener);
```

The *ATCommand.removeListener()* method removes a listener object that has been previously added to the internal list table of listener objects. After it has been removed from the list it will not be called when URCs occur. If it was not previously registered the list remains unchanged.

The same *ATCommandListener* may be added to several *ATCommand* instances and several *ATCommandListeners* may be added to the same *ATCommand.*

12.2 Programming the MIDlet

The life cycle and structure of MIDlets are described in Chapter 6. Since the MIDlets will run on Java ME™, all of Java ME™'s features, including threads, are available. Small applications, such as those without any timer functions or those used only for tests and simple examples, can be written without using threads. Longer applications should be implemented with threads.

12.2.1 Threads

Although small applications can be written without using threads longer applications should use them. The Java programming language is naturally multi-threaded which can make a substantial difference in the performance of your application. Therefore we recommend referring to Java descriptions on threads before making any choices about threading models. Threads can be created in two ways. A class can be a subclass of *Thread* or it can implement *Runnable*.

For example, threads can be launched in *startApp()* and destroyed in *destroyApp().* Note that destroying Java threads can be tricky. It is recommended that the developer read the Java documentation on threads. It may be necessary to poll a variable within the thread to see if it is still alive.

12.2.2 Example

```
/* This example derives a class from Thread and creates two instances
 * of the subclass. One thread instance finishes itself, the other one
 * is stopped by the main application. */

package example.threaddemo;
import javax.microedition.midlet.*;
public class ThreadDemo extends MIDlet {
  /* Member variables */
  boolean    runThreads = true; // Flag for stopping threads
  DemoThread thread1;           // First instance of DemoThread
  DemoThread thread2;           // Second instance of DemoThread

  /* Private class implementing the thread to be started by the
   * main application */
  private class DemoThread extends Thread {
    int loops;

    public DemoThread(int waitTime) {
      /* Store number of loops to execute */
      loops = waitTime;
      System.out.println("Thread(" + loops + "): Created");
    }

    public void run() {
      System.out.println("Thread(" + loops + "): Started");
      for (int i = 1; i <= loops; i++) {
        /* Check if main application asked thread to die */
        if (runThreads != true) {
          System.out.println("Thread(" + loops + "): Stopped from outside");
          /* Leave thread */
          return;
        }
```

```
        /* Print loop counter and wait 1 second,
         * do something useful here instead */
        System.out.println("Thread(" + loops + "): Loop " + i);
        try {
          Thread.sleep(1000);
        } catch(InterruptedException e) {
          System.out.println(e);
        }
      }
      System.out.println("Thread(" + loops + "): Finished naturally");
    }
  }
/**
   * ThreadDemo - constructor
   */
  public ThreadDemo() {
    System.out.println("ThreadDemo: Constructor, creating threads");
    thread1 = new DemoThread(2);
    thread2 = new DemoThread(6);
  }
/**
   * startApp()
   */
  public void startApp() throws MIDletStateChangeException {
    System.out.println("ThreadDemo: startApp, starting threads");
    thread1.start();
    thread2.start();
    System.out.println("ThreadDemo: Waiting 4 seconds before stopping threads");
    try {
      Thread.sleep(4000);
    } catch(InterruptedException e) {
      System.out.println(e);
    }
    destroyApp(true);
    System.out.println("ThreadDemo: Closing application");

  }
/**
   * pauseApp()
   */
  public void pauseApp() {
    System.out.println("ThreadDemo: pauseApp()");
  }
/**
   * destroyApp()
   */
  public void destroyApp(boolean cond) {
    System.out.println("ThreadDemo: destroyApp(" + cond + ")");
    System.out.println("ThreadDemo: Stopping threads from outsdide");
    runThreads = false;
    try {
      System.out.println("ThreadDemo: Waiting for threads to die");
      thread1.join();
      thread2.join();
    } catch(InterruptedException e) {
      System.out.println(e);
    }
    System.out.println("ThreadDemo: All threads died");
  notifyDestroyed();
  }
}
```