MULTOS SmartDeck v3.4.0.0 Developer's Reference Manual

Contents

Welcome To MULTOS SmartDeck

Welcome

Welcome to MULTOS SmartDeck, the complete application development system for MULTOS. SmartDeck allows you to use C and assembler to prepare applications that run on the MULTOS high security operating system.

This introductory chapter covers the contents of and conventions used in this manual: it should be read by everyone, but...

What is SmartDeck?

SmartDeck is a programming system which runs on MS Windows-based computers and is based around the Eclipse IDE. Programs which are prepared on these host machines using the tools in this package and are executed, not on the host, but on a MULTOS smart card or micro-controller.

C compiler

SmartDeck C is a faithful implementation of the ANSI and ISO standards for the programming language C. We have added some extensions that enhance usability in the MULTOS environment.

Because smart cards are small, memory-limited devices, we have to make a few concessions. For instance, the ANSI C input/output library is much too large for a smart card environment and a smart card has no way to display anything, so this implementation of C does not provide it.

Assembler

SmartDeck assembly language is a superset of the original MULTOS "MDS" assembly language. SmartDeck does not support the "MALT" assembly language syntax.

and more...

As well as providing cross-compilation technology, SmartDeck provides a PC-based simulation of MULTOS which is interfaced with the Eclipse IDE allowing you to debug your application quickly.

A set of tools for generating MULTOS standard application load units and a facility for loading and deleting applications on MULTOS cards provide the final stage of the software development lifecycle.

About this manual

There are five sections in this document:

- ***Getting Started Guide*** covers installing SmartDeck on your machine and setting up MULTOS projects within the Eclipse IDE.
- ***Tools Reference Guide*** contains detailed reference material about the SmartDeck tools.
- ***Codelet Development Guide*** *contains information on how to build and use Codelets.*
- ***Assembler User Guide*** contains detailed documentation covering how to use the assembler, the assembler notation, an instruction set reference, macros, and other assembler features and extensions.
- ***C User Guide*** contains documentation for the C compiler, including syntax and usage details and a description of the library functions supplied in the package.

What we don't tell you...

This documentation does not attempt to teach C programming or provide in-depth information on the Eclipse IDE. And similarly the documentation doesn't cover MULTOS or smart card application development in any great depth. We also assume that you're fairly familiar with the operating system of the host computer being used.

Getting Started Guide

Eclipse Installation

The MULTOS plugin for Eclipse has been developed for Eclipse CDT 2018-12 or later and JRE version 8 or greater. If you register to download SmartDeck from https://app.multos.com/ then these packages can be automatically installed for you using the Installation Manager.

GNU Tools

The plugin makes use of GNU *make.exe* and *rm.exe.* For convenience, versions of both these tools are included in the SmartDeck installation in the *bin* directory.

SmartDeck

MULTOS SmartDeck is distributed as a ZIP file. It does not require a hardware protection device and is not copy protected. If you are not using the provided Installation Manager, please install Eclipse before installing SmartDeck. Because Eclipse can be installed in any location, please manually copy the MULTOS plugin to the plugins folder in Eclipse.

Environment Variables

To make the tools available, you must add the **bin** directory, where the executable program files are installed, to your PATH variable. You do this by setting the "System Environment Variables" on your PC. Note that the Installation Manager installs SmartDeck to c:\program files (x86)\smartdeck by default and sets the environment for you.

Likewise, you must add the path to your **include** directory to the INCLUDE environment variable.

Installing the card component manually

The COM component we use to control the MULTOS card is registered by the installation program. Should you need to install the component manually, you can do so using the standard Windows program regsvr32. The command is:

```
regsvr32 /s htermlib.dll
```

The program will display a dialog box which tells you that the component installed correctly and that the registry has been altered to make the component known to client software. If you're curious, you can examine the interface to the SWT component using the Microsoft OLE viewer to decode the type library.

It is recommended to use the Installation Manager as this is handled automatically.

Eclipse Configuration

Once installed, you need to configure Eclipse to work with the MULTOS SmartDeck tools.

The **c:\temp\demo-workspace** folder is a completely set up, ready to use, Eclipse workspace containing an example application, **eloyalty.** You can immediately click on the project and click the debug icon. If you have installed SmartDeck and the other tools correctly, this will invoke the compiler, linker and debugger; the application will break on main(). **Note:** If SmartDeck is installed to anywhere other than

the default location (c:\program files (x86)\SmartDeck), you may need to edit the path to *hsim.exe* in the debugging.txt file.

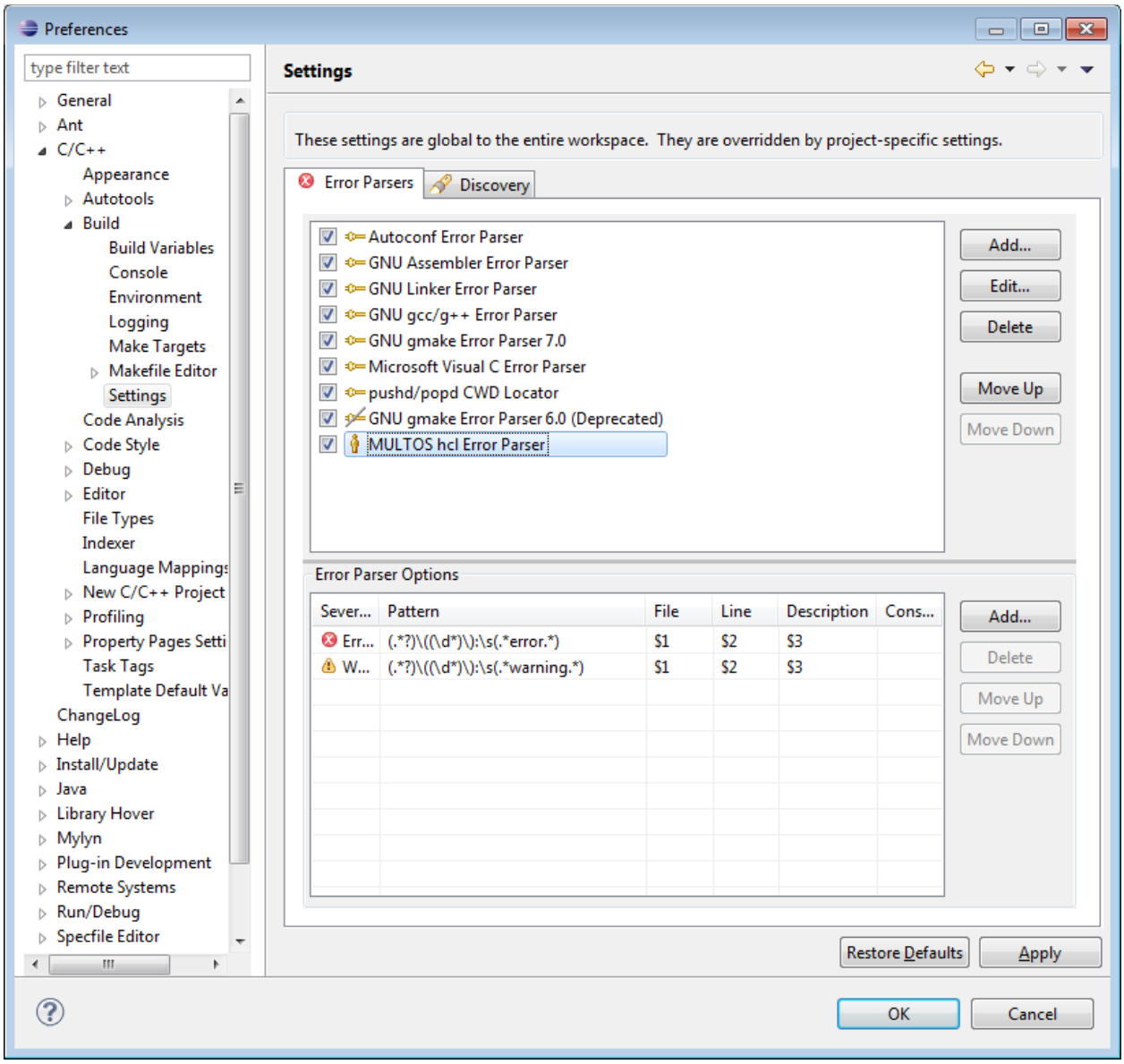
The rest of this section tells you how to set up a workspace and a new project from scratch.

Error Parsers

In order to be able to jump to error / warning locations in source code after compiling, you need to tell Eclipse how to parse the compiler output. These have to be manually defined in the workspace and the dialogue to use is **Window->Preferences->C/C++->Build>Settings**.

The regular expression to enter for error parsing is **(.*?)\((\d*)\):\s(.*error.*)** .
For warnings use **(.*?)\((\d*)\):\s(.*warning.*)** .

The screenshot below shows the setup.



Debugging Timeouts

Please be aware that the default debugger timeouts set in the Eclipse IDE may need to be lengthened. This can be done in **Window->Preferences->C/C++->Debug->GDB** (this option only appears

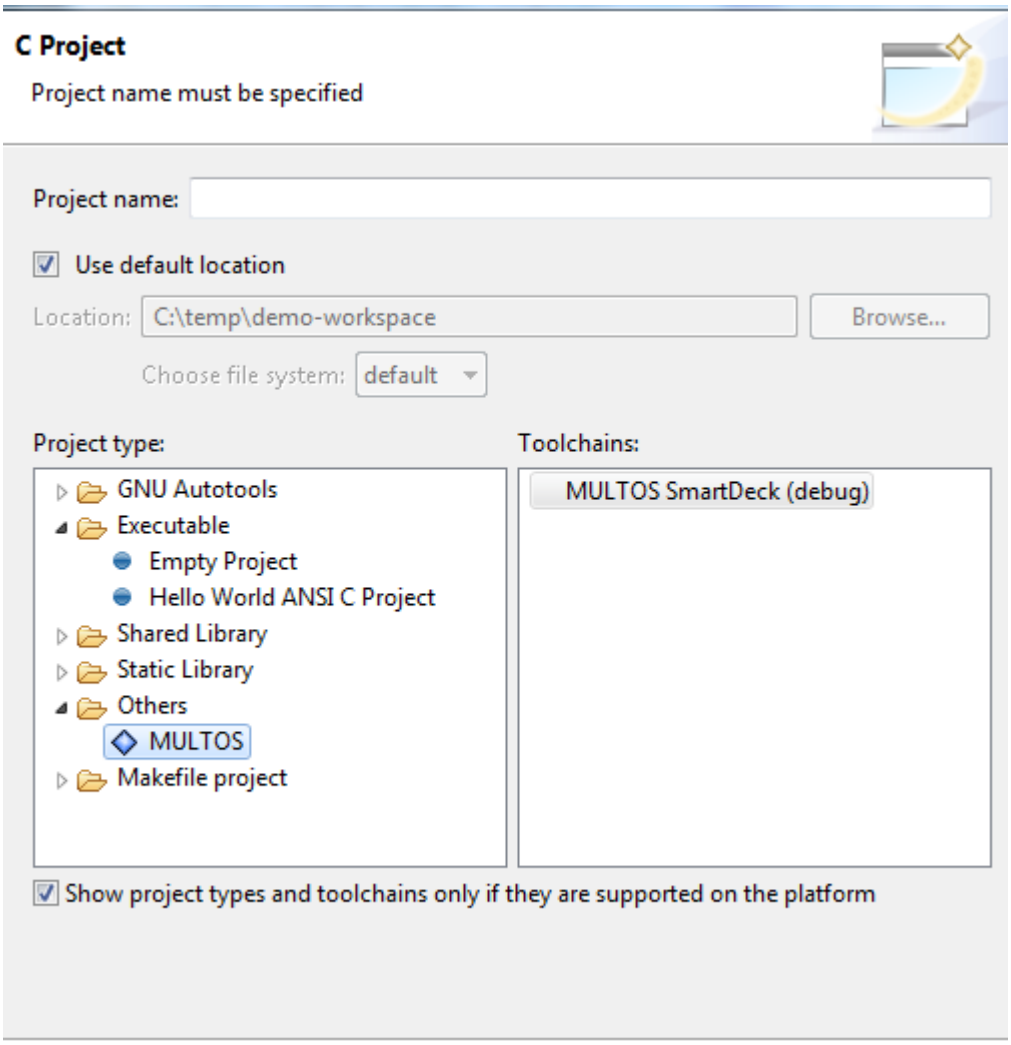
once you have created your first debugging config).

Disabling Unwanted CDT checking

The Eclipse CDT environment can report invalid syntax errors that don't match up with the actual compiler. To get rid of these unwanted messages uncheck all the checkboxes for **C/C++ Indexer Markers** in **Window->Preferences->General->Editors->Text Editors->Annotations**

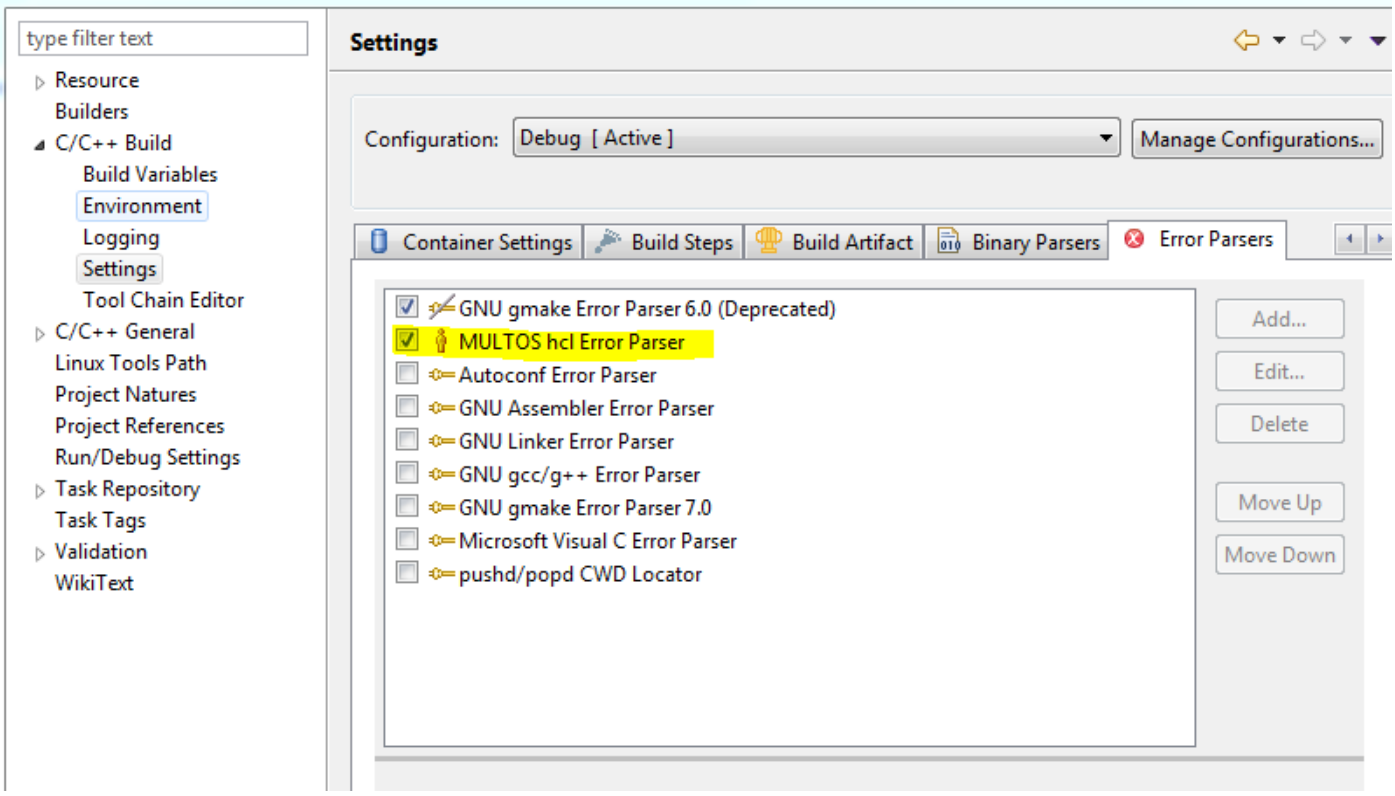
Creating a new Project in Eclipse

In the Eclipse Project Explorer , select **File->New->C Project**. In the dialogue presented type in a project name, select **MULTOS Project Type** and the *Debug* toolchain. Then click Finish.



Select Error Parsers

The globally created error parsers need to be selected for each project created. To do this, select your project in *Project Explorer* then select **Project->Properties->C/C++ Build->Settings** and tick the MULTOS error parser previously created.



Debugger Setup

This is slightly more complex to set up because of the flexibility that has been built in to the system.

Debugging Commands file

Firstly, in your project, create a text file called, for example, "debugging.txt". The first line of the file should be the full path to hsim.exe (or indeed any MULTOS simulator that supports the required interface). The second line should be the AID of the application to be debugged (in some simulators, not **hsim**, a comma separated list of AIDs can be provided).

Subsequent lines of the file are the command line parameters you wish to use with the simulator. For **hsim** these are the usual ones to select the application and send APDUs. For example, the file may look like this:

```
c:\program files (x86)\smartdeck\bin\hsim.exe
f3000003
-selectaid f3000003
; This is a comment as the first character on the line is a ;
-apdu 7001000002
; This is another comment.
-apdu 7002000002
```

You could include any of the other switches supported by **hsim**, but the name of the debug **.hzx** file is passed automatically, along with the switches required to support remote debugging.

The debugger sets the environment variable %CWD% to the root directory of the Eclipse project. %CWD% can be referenced from the debugging command file and from within any batch files

that the simulator may call (where applicable). The working directory of the debugger process (and hence the simulator) is set to this directory too.

If the application you are debugging delegates to other applications, then you must include the full path to the .hzx files of the delegate applications as a parameter to hsim.

By default, the debugger stops at the beginning of main() for every APDU being executed. You can prevent this behaviour by adding the following to the end of the debugging command file:

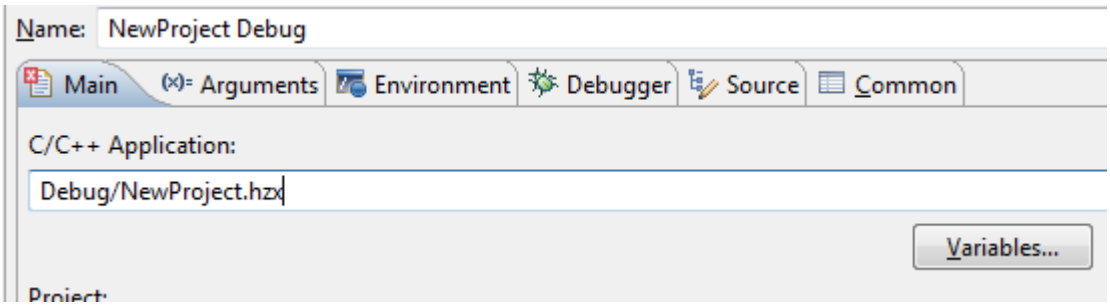
```
[OPTIONS]
NOBP
```

Eclipse Debugging Configuration

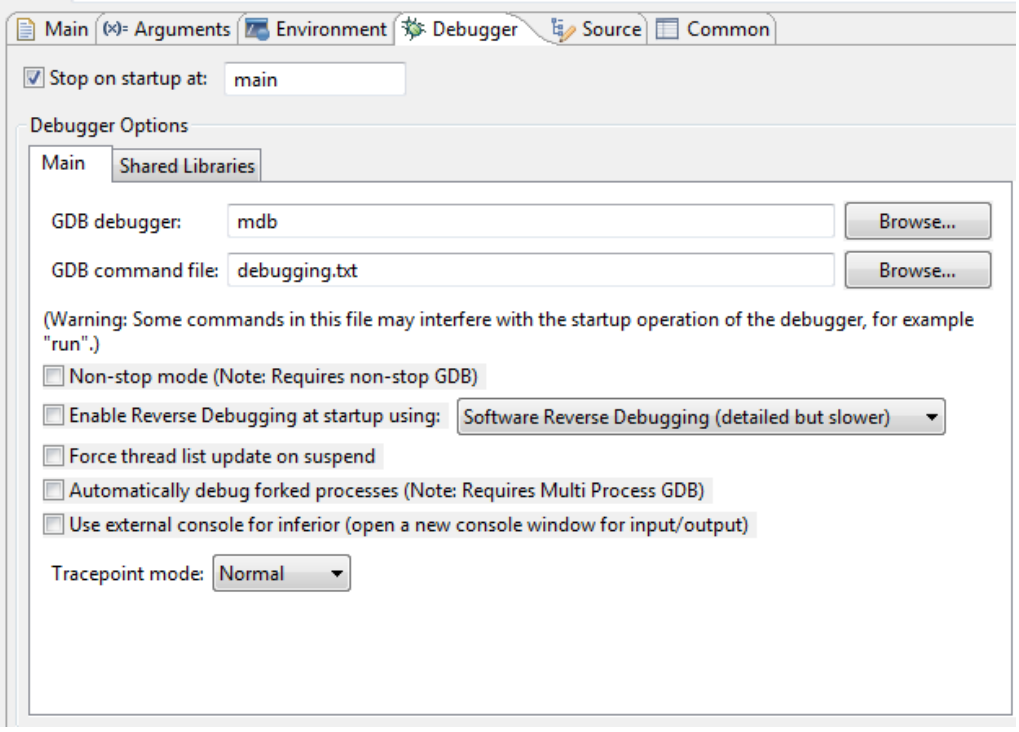
The last thing to do is to configure the debugger for your project. Select your project again in *Project Explorer* and then **Run->Debug Configurations** .

Under **C/C++ Application** create a new config using the 'New' button.

Fill in the **C/C++ Application** box with 'Debug/<projectname>.hzx'



On the debugging tab, change the GDB debugger field to **mdb** and select your debugging file created earlier as the GDB command file.

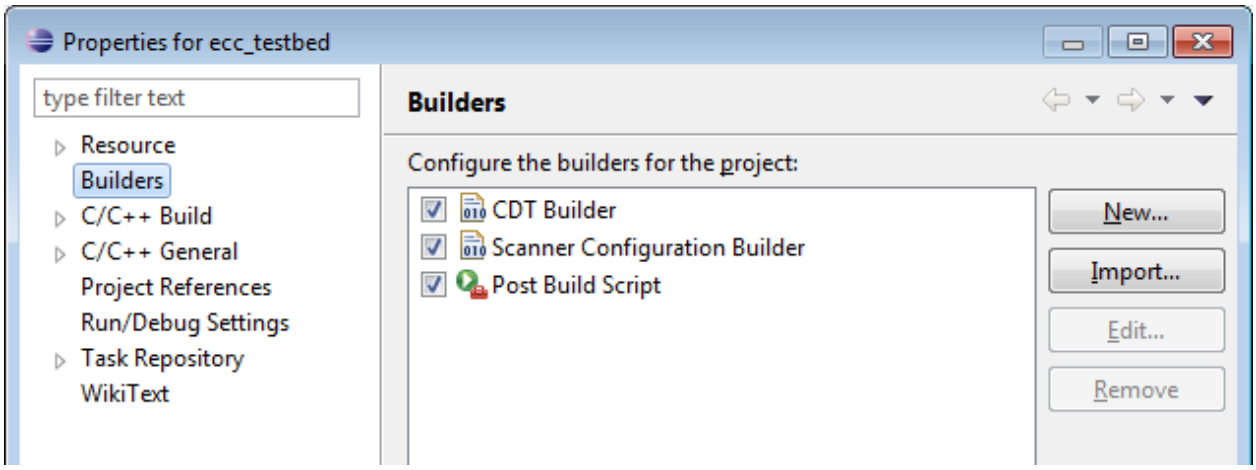


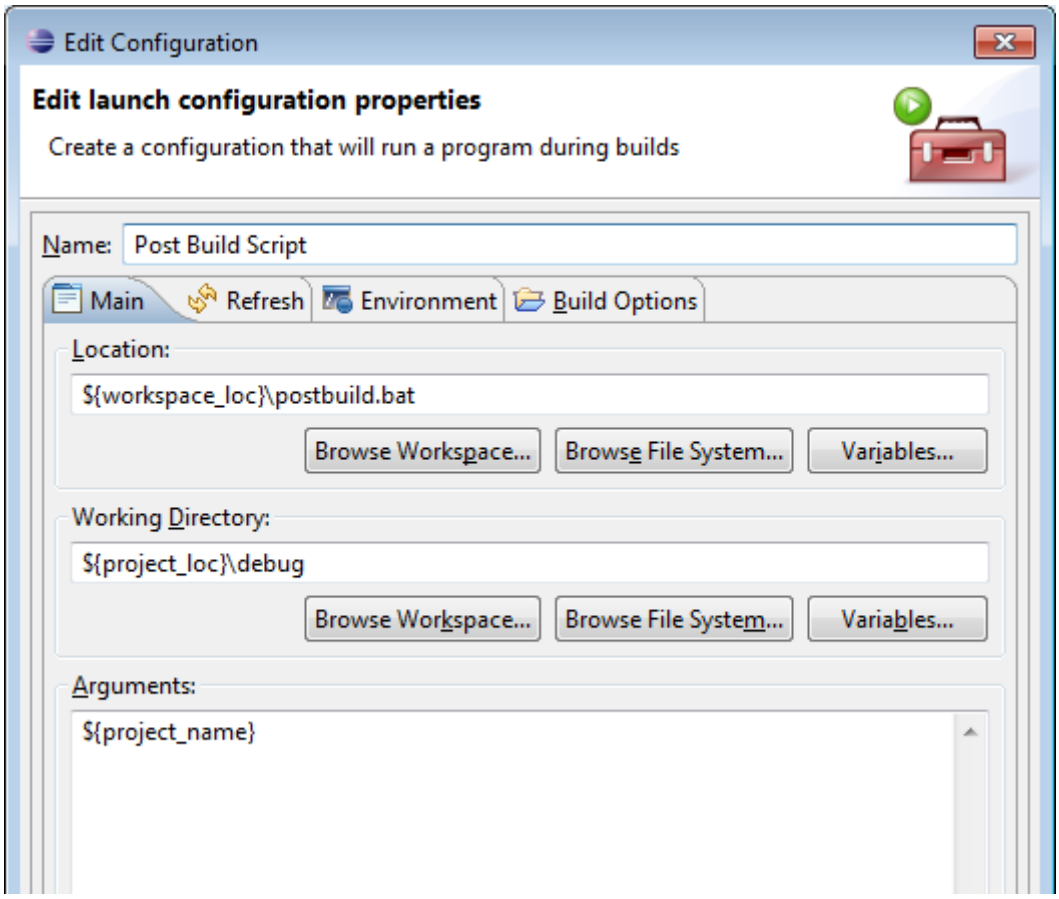
You are now ready to add 'C' source files and header files to your project.

Note: You can also copy then edit existing debug configurations, so once you've set up a debug config it is quicker for subsequent projects.

Post Build Scripts

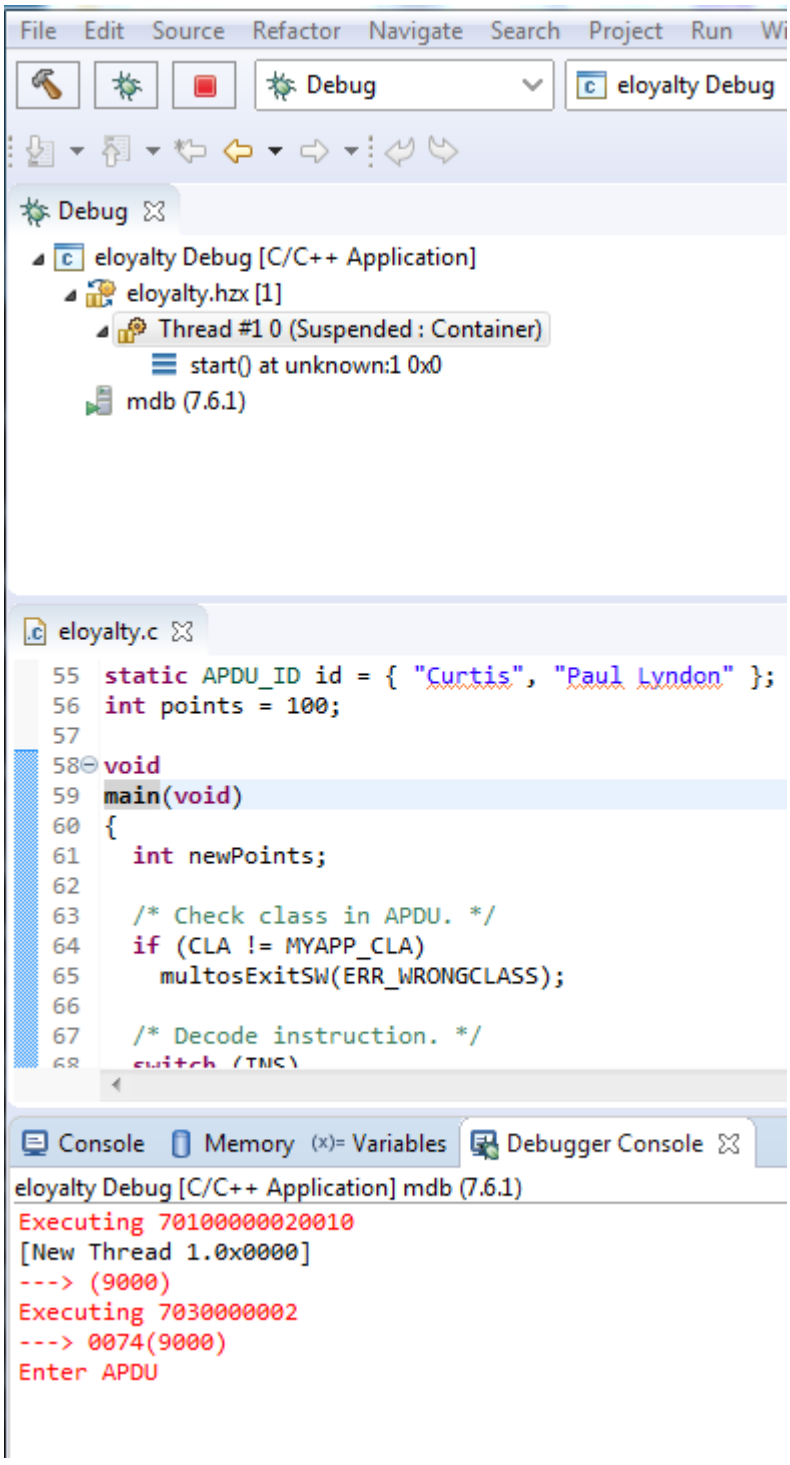
In addition to the standard build operations carried out by the MULTOS Toolchain, you may wish to add other steps such as generating an ALU and ALC using the halugen and melcertgen tools. A generic way to do this in Eclipse is to define a customer **Builder** for the project that is a generic script.





Executing APDUs

When using **hsim**, there are three methods of sending APDUs to the application. The first is by adding –apdu command line switches in *debugging.txt* as shown above. Once all these have been executed, or if none are specified, APDUs can be added via the Eclipse Debugger Console window asshown below:



Alternatively, APDUs may be sent to **hsim** from **hterm**. To start this, specify the –ifd switch in *debugging.txt* (see hsim command line options) and run **hterm** with the –sim and –interact switches (see hterm command line options).

When waiting for an APDU, the debugger is in a suspended state. After entering the

APDU and hitting the Enter key, it is necessary to click the *Resume* button (F8) to restart the debugger.

APDU responses are indicated with ---> . SW12 is shown in brackets.

Other simulators may use other mechanisms for application selection and APDU entry.

Supported Debugger Features

- APDU entry via a "script", the Debugger Console or hterm (for hsim)
- Step over and in
- Moving up / down the call stack
- Breakpoints
- Variable viewing (locals and parameters)
- Global viewing via *Expressions* window
- Hovering over a variable to get its value
- Viewing register values
- "View Memory" option for a variable
- Inclusion of **.asm** files (build and debug)
- Editing memory locations.
- Codelet development and debugging
- Debugging multiple loaded applications in one session (and delegation between them)
- Memory viewing
    - A literal address, e.g. 0x8010
    - A register, e.g. $DB
    - A pointer variable name
    - An address expression of the format **&(x)** or **&x** where x is a simple variable name (local or global)

Currently Unsupported Debugger Features

- Editing values
- Conditional watches / breakpoints
- Disassembly view

MULTOS basics

When a MULTOS card is inserted into an interface device (IFD), the IFD selects an application on the card and sends to it a series of commands to execute. Each application is identified and selected by its **application identifier** (AID). Commands are formatted and transmitted in the form of **application protocol data units** (APDUs). Applications reply to each APDU command with a status word indicating the result of the operation, and optionally with data.

Application design and architecture

Card applications do not differ from regular desktop applications in the way they are designed and developed. As with any software project, critical card applications should go through specification, design, and implementation, with validation and verification at each stage. Of course, there are different requirements for card applications, such as how security is managed, what to do if the card is torn from the reader, and so on, but the design process is unchanged.

Application identifiers

On the desktop, applications are held in files, and each file has a name. With smart cards, each application is identified and selected by an **application identifier** (AID). This naming convention is standardised by the International Standards Organisation and is defined in ISO 7816.

An AID is a sequence of bytes between 5 and 16 bytes in length. It consists of:

- *National registered application provider*. This is known as a RID, and its length is fixed at five bytes.
- *Proprietary application identifier extension*. This is known as a PIX, and its length varies between zero and 11 bytes.

ISO controls the assignment of RIDs, and each company manages the assignment of PIXs for AIDs. Each RID is unique, and you can obtain a RID for your company from ISO.

We'll pick a fictitious RID and PIX for our application's AID:

- RID: A0 01 02 03 04 (the required five bytes)
- PIX: 00 01 (two bytes)

About APDUs

APDUs are the way the card and terminal communicate; they are completely defined by ISO 7816. This section presents a brief overview of the various forms of APDUs we can use.

Each APDU exchanged between the card and the terminal is composed of a command APDU and a response APDU, usually abbreviated as C-APDU and RAPDU, respectively. The command APDU is the command sent to the card, and the response APDU returns to the terminal the execution result of the command. Command APDUs and response APDUs are exchanged alternately between a card and the IFD.

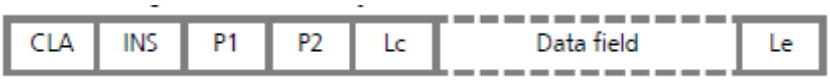
It is important to stress that smart cards never initiate communications, they only respond to command APDUs sent to them by the IFD.

APDUs and MULTOS

MULTOS receives all commands sent by the terminal. When MULTOS receives a command, it decides whether the command should be handled by MULTOS itself or whether the command should be passed to an application on the card.

We'll gloss over the details of how this decision is made, for now, and just introduce the notion of a **currently selected application**. This application receives all the commands MULTOS doesn't process itself; it is the application's responsibility to act on them and to return a response.

A command APDU consists of a four-byte mandatory header followed by a variable- length conditional body. The structure of the command APDU is:



The fields within the command APDU are:

- **CLA — Class of instruction.** A mandatory single byte that indicates the structure and format for a category of command and response APDUs.
- **INS — Instruction code.** A mandatory single byte that specifies the instruction of the command.
- **P1 and P2 — Instruction parameters.** Two mandatory single-byte parameters that provide qualifications to the instruction.
- **Lc — Length of command data.** An optional single byte indicating the number of bytes present in the data field of the command*.
- **Data field.** An optional sequence of Lc bytes in the data field of the command.
- **Le — Length of expected response.** An optional single byte indicating the maximum number of bytes expected in the data field of the response to the command*.

A response APDU consists of a variable-length conditional body and a two-byte mandatory trailer as follows:

The fields within the response APDU are:

- **Data field.** An optional sequence of bytes received in the data field of the response.
- **SW1 and SW2 — Status words.** Two mandatory single-byte values that denote the processing state in the card.

***Note:** SmartDeck also supports long form APDUs for T=1 and T=CL devices.

Application selection

An application on a MULTOS card is inactive until it is explicitly selected with the SELECT command. When MULTOS receives a SELECT command, it searches for the application whose AID matches the one specified in the command. If a match is found, MULTOS prepares the application to be selected and, if an application is already selected, MULTOS deselects it.

Once an application is selected, MULTOS forwards all subsequent APDU commands (including further SELECT commands) to the application. In the main method, the application interprets each incoming command APDU and performs whatever is requested by the command. For each command APDU sent, the application responds by sending back a response APDU.

This command-and-response dialogue continues until a new application is selected or the card is removed from the IFD. When deselected, an application becomes inactive until the next time it is selected.

Programming Topics

This section describes special features of SmartDeck and MULTOS that you'll need to know about when you start writing applications.

Attributes

The C compiler and assembler support attributes that allow you to easily set up application parametrers. These attributes are interpreted by the SmartDeck tools to provide extra features.

Attribute syntax in C

An attribute is specified using the syntax:

```
#pragma attribute("attribute name", "attribute value")
```

The C compiler inserts the attribute into the object file, and the linker gathers the attributes of all linked modules and places them into the linked executable.

The compiler and linker do not interpret the attributes or ascribe a meaning to them. You can use attributes to insert copyright notices into object files, for example. Attributes are not placed into the code of the application, they merely act as flexible additional information for processing by the SmartDeck tools.

Setting attributes on the command line

If you don't want to place attributes in your source code, you can give them on the `hcl` command line as follows:

```
hcl -Aattribute=value ...
```

This instructs the linker to set the attribute to the given value. An attribute given on the command line in this way overrides any attribute that you have set in a source file.

As an example, you can insert a copyright notice into `eloyalty.hzx` when compiled from `eloyalty.c` using the following:

```
hcl -Acopyright=MyCompany eloyalty.c
```

If you wish to include spaces in the attribute, you must place quotation marks around the option like this:

```
hcl "-Acopyright=Copyright (c) 2006 MyCompany Inc" eloyalty.c
```

Listing attributes

You can list the attributes of an object file or an executable by using the `-A` switch of the object lister `hls`. For example, the following will list the attributes that are present in the linked executable `eloyalty.hzx`:

```
hls -A eloyalty.hzx
```

Attributes interpreted by SmartDeck

The SmartDeck ALU output routines and the SmartDeck loader interpret attributes to set application parameters when loading the application onto a card.

Setting the application ID (AID)

```
#pragma attribute("aid", "hex-string")
```

This sets the application ID for the application to **hex-string**, which must be a hex- encoded string. For example, the following sets the AID of the application to F1 00 00 00 00:

```
#pragma attribute("aid", "F1 00 00 00 00")
```

If you prefer, you can omit the spaces and use:

```
#pragma attribute("aid", "F100000000")
```

or even:

```
#pragma attribute("aid", "F1 0000 0000")
```

When the application is loaded onto the card using `hterm`'s automatic certificate generation, its application ID is set automatically to this AID.

Setting the file control information (FCI)

You can set the file control information record using the same syntax:

```
#pragma attribute("fci", "hex-string")
```

The file control information is returned by MULTOS by the Get File Control Information primitive.

Setting the file directory entry (DIR)

And set the directory file entry using:

```
#pragma attribute("dir", "hex-string")
```

Alternatively, the attribute "name" can be specified and this will be automatically combined with the "aid" attribute to form the "dir" attribute value. This is easier than having to know how to format the DIR entry.

```
#pragma attribute("name", "string")
```

Setting the ATR type (ATRTYPE)

The ATR type to be either "primary" or "secondary"

```
#pragma attribute("atrtype", "primary")
```

Specifying that the application controls the ATS

This is interpreted when creating an Application Definition File with *melcertgen*

```
#pragma attribute("atscontrol")
```

Specifying a shell application

You can specify that you application is to be a shell application using

```
#pragma attribute("shell")
```

Specifying a default application

You can specify that you application is to be a default application using

```
#pragma attribute("default")
```


Setting the access_list

The access list to be used when loading from the .hzx file or creating a default ALC. It must be a two byte value.

```
#pragma attribute("access_list", "hex-string")
```

Differences between MULTOS implementations

Some features of MULTOS are designated **optional** which means that a MULTOS implementation may or may not support that particular feature. Testing your code on the card you are deploying onto, therefore, is vital because the SmartDeck MULTOS simulator supports **all** optional primitives.

The features that a card supports are described in the **MULTOS Implementation Report** document which is available fon the MAOSCO web site.

Session data and stack sizes

The size of dynamic data varies between MULTOS implementations, so you should check that your application works on the card that you wish to deploy onto. To check the amount of session and dynamic data available on a card, consult the **MULTOS Implementation Report**.

Optional support for MULTOS N and V flags

The N and V flags of the MULTOS condition code register are designated as optional flags. The SmartDeck compiler generates code that does not rely on the MULTOS card supporting signed arithmetic and as such does not use the N and V flags. All library routines are written so that they will execute correctly on cards which do not support signed arithmetic.

You can safely deploy your code on any card whether it supports signed arithmetic or not, fully confident that it will work correctly.

Optional MULTOS primitives

Some MULTOS primitives are designated optional, such as certain crytographic primitives and transaction protection. It is impossible for the compiler to check that the primitives you use are available on the card you deploy onto. If you use a primitive that is not supported by the card, your application will abend.

Atomicity and data item protection

You cannot rely on the code generator of the compiler to write atomically to a variable.

As an example, consider the following program fragment:

```
static long var;
void inc_var(void)
{
  var += 2;
}
```

The compiler uses two instructions to increment the variable:

```
INCN SB[var], 4

INCN SB[var], 4
```

If you want to use atomic writes and data item protection you should always check the code generated by the compiler for your critical routines. Rather than using data item protection, you should consider using transaction protection which is easier to use than data item protection in high-level languages such as C.

The compilers and MULTOS user flags

The SmartDeck compilers and runtime system do not use the four flags in the condition code register set aside for the application to use. You may safely use these flags in assembler without having them changed by compiler code or the runtime system.

Modularity

The SmartDeck assembler writes object files which can be linked together with other units to form an application. This is quite unlike the MDS assembler where all source code is compiled as a single unit.

The C compiler

The SmartDeck C compiler is an implementation of ANSI standard C. When using the optimization features of the linker, SmartDeck C makes an ideal development system for smart card applications.

The simulator and debugger

The SmartDeck debugger provides allows you to debug C and assembler programs at the source level.

Troubleshooting

Simulator and smart card problems

Application works on the simulator but fails on the card

This is a common problem when developing smart card applications. In fact, it happens when using any simulation environment, and isn't just a problem with the SmartDecktools.

There are lots of reasons why your application will run on the SmartDeck simulator but not on the card, and here we present a set of steps to follow to rule out some common problems. However, these may not always be enough to figure out why your card application doesn't work — in which case, you can always ask our support department for further suggestions and help.

- Check your resource use

The simulator is rich in resources, with lots of room for code and data. In particular the dynamic and session data are large. It could be that your application will not work on the card because you use too much session or dynamic data, in which case the application will abend. Trying to figure out if this is the case will be difficult. However, what you can do is restrict the amount of dynamic data in the simulator by using the `-ds` switch to define the amount of dynamic data available to applications. You can find the amount of dynamic data your card has from MAOSCO; then, plug the value into the simulator and see whether your program still works. If it doesn't, the debugger can tell you where you used too much dynamic data.

- Check your card

You've used MULTOS features which are available in the simulator but not on your card.

Application works on the card but fails on the simulator

This should never happen as we intend the simulator to be a faithful implementation of a MULTOS smart card. If this does happen then it is very likely to be a problem in our simulator, and you should report this to MAOSCO so they can investigate.

Manual installation of the card component

The COM component we use to control the MULTOS card is registered by the installation program. Should you need to install the component manually, you can do so using the standard Windows program `regsvr32`. The command is:

```
regsvr32 /s htermlib.dll
```

The program will display a dialog box which tells you that the component installed correctly and that the registry has been altered to make the component known to client software

SmartDeck Components List

SmartDeck comprises a number of separate programs that work together via proprietory files to enable you to develop and test MULTOS applications. There are three different file formats in SmartDeck:

- Object files (`.hzo`) that contain compiled program code
- Library files (`.hza`) that are collections of object files. Library files are stored in the popular Zip file format.
- Executable files (`.hzx`) that is a fully linked executable program.

The contents of object and executable files can be listed using the `hls` program. The contents of library files can be viewed using the archiver or a Zip file viewer.

SmartDeck object files can be created:

- from assembly language source code using the has program.
- from C source code using the hcc program.
- Library files are created using the `har` program.
- Object files and library files are linked together using the `hld` program to create an executable file.

Executable files can be loaded and debugged on the MULTOS debugger/simulator programs `mdb` and `hsim`.

Executable files can also be loaded onto MULTOS cards using the `hterm` program.

MULTOS application load units can be generated from executable files using the `halugen` program.

The use of the C compiler, assembler, linker, and even the archiver is coordinated by the compiler driver, `hcl.exe` so you won't need to use these programs directly.

Each of these programs is described by a chapter in this manual. The following table lists these programs, with references to their primary documentation sections.

NOTE: Most of the time you will execute these applications from within the Eclipse IDE. In fact, the standard MULTOS debug toolchain takes care of compiling, linking and debugging your applications. The following chapters should be refered to when you wish to change the default settings or add further processing steps.

Program	Description
hcl.exe	**Compiler driver**: provides a useful way to get files compiled, assembled and linked
hcc.exe	**C compiler**: compiles modules written in C
mdb.exe	**Eclipse gdb/mi debugger**: Provides the debugging interface between Eclipse and the MULTOS simulator.
has.exe	**MULTOS Assembler**: assembles modules written in assembly language
hld.exe	**Linker:** Required for linking compiled and assembled files, along with run-time startup code and support libraries, into formats suitable for downloading or debugging
hsim.exe	**MULTOS Simulator:** used in conjunction with **mdb** but can also be used stand-alone.
hterm.exe	**Loader:** Used to load and delete application from MULTOS cards
har.exe	**Archiver:** Consolidates multiple object files into a single, object code library
hls.exe	**Object file lister**: displays useful information held in unlinked files and linked executables.
hkeygen.exe	**RSA key pair generator**: creates a private and public RSA key pair, suitable for use with MULTOS cryptography primitives
halugen.exe	**ALU generator**: creates a standard MULTOS application load unit.
melcertgen.exe	**ALC/ADC generator**: creates load and delete certificates for developer cards.
meldump.exe	**MULTOS file list**: outputs contents of standard MULTOS files.
hex.exe	**Extractor utility**: used to prepare images in various formats

Table 6 Components of MULTOS SmartDeck

Compiler Driver Reference

This section describes the switches accepted by the SmartDeck compiler driver, `hcl`. The compiler driver is capable of controlling compilation by all supported language compilers and the final link by the SmartDeck linker. It can also construct libraries automatically.

In contrast to many compilation and assembly language development systems, with SmartDeck you don't invoke the assembler or compiler directly. Instead you'll normally use the ***compiler driver*** `hcl` as it provides an easy way to get files compiled, assembled, and linked. This section will introduce you to using the compiler driver to convert your source files to object files, executables, or other formats.

We recommend that you use the compiler driver rather than use the assembler or compiler directly because there the driver can assemble multiple files using one command line and can invoke the linker for you too. There is no reason why you should not invoke the assembler or compiler directly yourself, but you'll find that typing in all the required options is quite tedious—and why do that when `hcl` will provide them for you automatically?

File naming conventions

The compiler driver uses file extensions to distinguish the language the source file is written in. The compiler driver recognises the extension `.c` as C source files, `.s` and `.asm` as assembly code files, and `.hzo` as object code files.

We strongly recommend that you adopt these extensions for your source files and object files because you'll find that using the SmartDeck tools is much easier if you do—as you'll see later.

C language files

When the compiler driver finds a file with a `.c` extension, it runs the C compiler to convert it to object code.

Assembly language files

When the compiler driver finds a file with a `.s` or `.asm` extension, it runs the assembler to convert it to object code.

Object code files

When the compiler driver finds a file with a `.hzo` extension, it passes it to the linker to include it in the final application.

Translating files

Translating a single file

The first thing that you'll probably want to do is assemble a single source file to get an object file.

Suppose that you need to assemble the file `test.asm` to an object file— you can do this using `hcl` as we suggested before, and all you type is:

```
hcl -c test.asm
```

The compiler driver invokes the assembler with all the necessary options to make it produce the object file `test.hzo`. The option `-c` tells `hcl` to assemble `test.s` to an object file, but to stop there and do nothing else.

To see what actually happens behind the scenes, you can ask `hcl` to show you the command lines that it executes by adding the `-v` option: `hcl -v -c test.asm`

On my computer, the commands echoed to the screen are:

```
F:\SmartDeck\Examples>hcl -v -c test.asmF:\SmartDeck\bin\has -JF:\SmartDeck\include -
I. test.s -o test.hzo
```

Now you can see why using the compiler driver is so much easier! It doesn't matter about the order of the `-v` and `-c` switches, nor does it matter where you place them on the command line.

Assembling multiple files

Where the compiler driver really shines is that it can assemble more than one file with a single command line. Extending the example above, suppose you wish to assemble the three files `test.asm`, `apdu.asm`, and `mem.asm` to three object files; this is no problem:

```
hcl -c test.asm apdu.asm mem.asm
```

The compiler driver invokes the assembler three times to process the three source files. This sleight of hand is revealed using `-v`:

```
F:\SmartDeck\Examples>hcl -v -c test.asm apdu.asm handler.asm
F:\SmartDeck\bin\has -JF:\SmartDeck\include -I. test.asm -o test.hzo
F:\SmartDeck\bin\has -JF:\SmartDeck\include -I. apdu.asm -oapdu.hzo
F:\SmartDeck\bin\has -JF:\SmartDeck\include -I. mem.asm -o mem.hzo
```

Command syntax

You invoke the compiler driver using the following syntax:

```
hcl [ option | file ]...
```

Files

file is a source file to compile. The compiler driver uses the extension of the file to invoke the appropriate compiler. The compile driver supports the following file types:

Extension	Compiler invoked
`.c`	C compiler, hcc.
`.s .asm`	Assembler, has.
`.hzo`	None, object file is sent to linker, hld

Table 7 Languages and file extensions

- ### Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The compiler supports the following command line options:

Option	Description
`-A`***name*** `[=`***val***`]`	Define the attribute ***name*** and optionally set it to ***val***.
`-ansi`	Warn about potential ANSI problems

`-ar`	Create an archive library as output, do not link
`-c`	Compile to object code, do not link
`-D`***name***	Define the pre-processor symbol ***name*** and optionally initialise it to **val**
`-D`***name***`=`***val***	
`-E`***name***	Set program entry symbol to ***name***
`-F`***fmt***	Set linker output format to ***fmt***
`-g`	Generate symbolic debugging information
`-help`	Show tool help page
`-I-`	Don't search standard directories for include files
`-I`***dir***	Add ***dir*** to the end of the user include search list
`-J`***dir***	Add ***dir*** to the end of the system include search list
`-L`***dir***	Set the library search directory to ***dir***
`-l`***x***	Search library ***x*** to resolve symbols
`-M`	Display linkage map on standard output
`-n`	Dry run—do not run compilers, assembler, or linker
`-O`	Optimise output
`-opt` *n*	Use optimised instruction group 'n'. See [MDRM] for details.
`-nocen`	Do not issue Compact Enable instruction. See [MDRM] for details.
`-o` ***file***	Leave output in ***file***
`-pt` ***file***	File containing the linker patch table for patchable codelet functions
`-pure`	Check that the application has no static data
`-r`	Do a partial link.
`-s-`	Do not link the standard startup file `crt0.hzo`.
`-s`***name***	Use startup file ***name*****`.hzo`**
`-T`***name***`=`***l-addr***[`/`***raddr***]	Set section ***name*** to load at ***l-addr*** and run at ***r-addr***
`-U`***name***	Undefine pre-processor symbol ***name***
`-v`	Show commands as they are executed
`-V`	Display tool versions on execution
`-w`	Suppress warnings
`-W[abcfjtl]`***arg***	Pass *arg* to specified tool a – assembler b – basic compiler c – C compiler f – FORTH compiler l – linker
`-X[abcfjtl]`*x*	Use *x* as translator for given language – the letters have the same meanings as the `-W` option

Table 8 Compiler driver command line options

Verbose execution

The SmartDeck compiler driver and compilers usually operate without displaying any information messages or banners—only diagnostics such as errors and warnings are displayed.

If the `-v` switch is given, the compiler driver displays its version and it passes the switch on to each compiler and the linker so that they display their respective versions.

The `-v` switch displays command lines executed by the compiler driver.

Finding include files

In order to find include files the compiler driver arranges for the compilers to search a number of standard directories. You can add directories to the search path using the `-I` switch which is passed on to each of the language processors.

Macro definitions

Macros can be defined using the `-D` switch and undefined using the `-U` switch. The macro definitions are passed on to the respective language compiler which is responsible for interpreting the definitions and providing them to the programmer within the language. The `-D` switch takes the form:

`-D`***name***

or

`-D`***name***`=`***value***

The first defines the macro ***name*** but without an associated replacement value, and the second defines the same macro with the replacement value ***name***. The `-U` switch is similar to `-D` but only supports the format

`-U`***name***

C compiler

When passed to the C compiler, the macros are interpreted by the compiler's pre- processor according to the ANSI standard. The pre-processor interprets a macro defined using `-D`***name*** as equivalent to the declaration:

`#define` ***name***

A macro defined using `-D`***name***`=`***value*** is equivalent to: `#define` ***name value***
A macro undefined using `-U`***name*** is equivalent to `#undef` ***name***

Linking in libraries

You can link in libraries using the `-l`x option. You need to specify the directory where libraries are to be found, and you can do this with the `-L` option.

- Example

Link the library libcrypto.hza and libio.hza found in \SmartDeck\lib:

```
hcl -L\SmartDeck\lib -lcrypto -lio
```

Building archives

The compiler driver can be used to simplify building archive libraries. For example three source source files can be compiled and a library made with the following command

```
hcl -ar -o mylib.hza file1.c file2.c file3.c
```

Creating ALU's

Unprotected ALU's can be simply created using the `-Falu` option. To create protected and confidential ALU's use the `halugen` program

The Simulator

You can use the MULTOS debugger to test applications during development rather than trying to debug applications on a real smart card. The simulator is a faithful implementation of the MULTOS virtual machine, including:

- interpretation of all MEL instructions
- implementation of MULTOS memory and application structures
- support for all MULTOS primitives
- full-strength cryptography and delegation
- MULTOS and application APDU commands
- a `printf` extension primitive to help debug in difficult situations
- simulated command interface enabling host program to send commands to the simulator

The simulator can be run either in standalone mode or under control of the Eclipse IDE.

Command syntax

```
hsim [ options ]... file1 file2 .. filen
```

Command arguments can be put into a command file and referenced by preceding the filename with a `@`

```
hdb @commandfile
```

Files

***file*$_1$, *file*$_2$ .. *file*$_n$** are the applications (executables produced by the linker) that are loaded into the simulator.

Options

options are command-line options. Options are case sensitive and cannot be abbreviated.

Option	Description
`-apdu` **"*command*"**	Stack an APDU command. A command is a sequence of twodigit, space-separated, hexadecimal numbers. This option can be used several times to define a sequence of APDU commands.
-addStatic ***blocks***	Number of additional 255 byte blocks of static memory to make available above that allocated in the application source code. See [MDG] for details of "Additional Static" memory.
-cardtype type	Specify the cardtype one of HitachiV3, HitachiV4, KeycorpInfineonV4, KeycorpInfineonV4P, MI-M3, MI-M4, MI-M5, ersa192, ersa256. This option is used to select development card specific cryptography keys (such as that used in AHash).
`-ds` ***size***	Define the size of the dynamic region using a C-style number (default is based on the cardtype).

-event ***id***	Simulate the raising of a Process Event with the given ID (default value is 0).
-ifd ***name***	When this option is used APDU's can be supplied to the simulator fromthe SmartDeck Terminal COM automation component.
`-init` ***sym***=***val***	Redefine the initialisation of a global static variable. The value is a string of hexadecimal numbers. If the symbol is a pointer then the locations pointed at are updated.
`-noclearpublic`	When this option is used the public memory area will not be zeroed between APDUs. The default behaviour is to zero the public memory area between APDUs.
`-ps` ***size***	Define the size of the public region using a C-style number (default is based on the cardtype)..
`-select name`	Select named application to run.
`-selectaid aid`	Select the application to run by its hex application identifier value.
`-t=0`	Interpret APDU's in T=0 protocol. This means that the debugger/simulator treat case 3 and case 4 commands the same (an Le byte isn't needed for case 4). For a case 4 command the debugger/simulator sends a GetResponse status.

Table 9 – Simulator option summary

The following options may be useful for standalone use of the simulator:

Option	Description
`-count`	Display the instruction count on exit.
`-e`	Exit on error (default is to report error and continue from the next instruction).
`-p`	On exit files are written with instruction count information. The output files have the same names as the .hzx input files with the suffix "`_p`" added.
`-pn`	On exit files are written with instruction count information. The output files have the same names as the .hzx input files.
`-t`	Trace instructions.
`-log`	Log APDUs sent.

Table 10 Simulator additional options

Example

```
hdb app1.hzx app2.hzx app3.hzx -select app2 -apdu "70 10 00 0000"
```

This example loads three applications, selects one of the applications and sends it the command "`70 10 00 00 00`".

Example

```
hsim eloyalty.hzx -ifd sim
```

This example starts a standalone simulator that can communicate with the

SmartDeck Terminal COM automation component for example

```
hterm -sim sim -select eloyalty -apdu "70 30 00 00 02"
```

Example

```
hsim eloyalty.hzx -select eloyalty -apdu "70 30 00 00 02" -p
```

This example loads an application, selects it, sends an apdu to it and then exits. On exit the file `eloyalty.hzx_p` is written which contains both the application and instruction count information on the program run. The command line

```
hsim eloyalty.hzx_p -select eloyalty -apdu "70 30 00 00 02" -pn
```

will load the application (now containing the instruction counts from the last run), select it, send an APDU to it and then rewrite the file `eloyalty.hzx_p`with the accumulated instruction counts of both program runs.

Simulation status

The simulation can be either running (executing instructions) or stopped. The simulation is stopped by

- hitting a breakpoint
- completing a single step
- pressing the stop button
- the application executing an instruction that causes a MULTOS abend
- returning control to MULTOS by executing an EXIT instruction

When the simulation is stopped, you can examine its state using the debugger.
When the simulation is restarted after it has executed a MULTOS exit instruction, it starts executing at the first instruction of the selected application. Session and static data values are preserved, allowing you to test the behaviour of your application using a series of APDUs be tested.

The simluator will not run until an application has been selected. This can be done on the command line with the `-select` option. If an application is built with the shell or default attribute then it is automatically selected.

The simulator will not run until it has received an APDU to process. APDU's can be provided on the command line (-apdu option, can be used multiple times), typed into the Eclipse console window or can be supplied programmatically using the `-ifd` option together with a suitable host program.

Static memory can be preserved between runs of hsim. To enable this, create an empty **.mem** file in the current working directory (the project directory when executed via Eclipse). The name of the file should be the application ID. e.g. **f3000001.mem** . The file will be updated every time that static memory is written to. Subsequent runs of hsim will load the state from the file as it was left after the previous run.

Working with Cards

To assist you when testing your application on a MULTOS card or debugging your application on the SmartDeck simulator, we've provided the "SmartDeck Terminal COM automation component." We refer to it here simply as the **SDT component**.

Using this component, you can:

- Load an application onto a card.
- Delete an application from a card.
- List all applications present on a card.
- Select applications and send APDUs to them.
- Retrieve MULTOS and manufacturer data for a card.

The SDT component supports interaction either with a PC/SC smart card reader or with the SmartDeck Simulator/Debugger. This enables you to run identical test cases on both.
Because this is a COM component, you can use it in your own PC programs or scripts; we show how to do this in " Using the SDT component". For now, we'll concentrate on using the `hterm` utility, which makes it easy to manage applications on your development cards and provides a simple facility to send APDU commands.

Managing card applications using hterm

The program `hterm` provides basic access to the facilities of the SDT component. More-advanced use of the SDT component can be achieved programmatically, for example by using a suitable scripting language.

Before we load applications onto the card, it's worth checking that the card, card reader, and software work together properly. First check that PC/SC is up and running by typing

```
hterm -readers
```

This should return a list of PC/SC reader names for smart card readers that you have installed on your computer. If you have just one reader then `hterm` will always connect to this as the default reader (the subsequent documentation examples assume this). If you have more than one PC/SC reader then you'll need to specify which reader to use by using the `-pcsc` option.

Now insert a MULTOS card into the card reader and type:

```
hterm -atr
```

For a new MULTOS developer card, you should see something like this:

```
ATR 0x3b 0x6f 0x00 0x00 0x80 0x31 0xe0 0x6b 0x84 0x03 0x03 0x04 0x05 0x55 0x55 0x55
0x55 0x55 0x55
```

This displays the Answer To Reset (ATR) from the card. The exact values of the bytes may differ, but the command should execute without problems. If you have problems, please check that your reader is working using the vendor-supplied testing programs.

Now for something more adventurous. Type:

```
hterm -card
```

This should (depending on the type of MULTOS card you have) return one of HitachiV3, HitachiV4, KeycorpInfineonV4 or KeycorpInfineonV4P.

If you can get this far, your card reader and software is working well. Now we can use `hterm` to load applications onto a card and do some simple testing.

The eloyalty application that is supplied with the toolset can be loaded as follows

```
hterm -load eloyalty.hzx
```

You can verify that it has been loaded by examining the MULTOS directory with the command

```
hterm -dir
```

Which will detail the application label and the application identifier that the SDT component has allocated to it. You can now select the application, and send two APDU commands:

```
hterm -select eloyalty -apdu "70 30 00 00 02" -apdu "70 10 00 00 0200 10"
```

This will show the apdu commands and their reponses on the console. You can now delete the application from the card, this can done either by

```
hterm -delete eloyalty
```

or all applications can be deleted using

```
hterm -clean
```

Loading and deleting applications

The SDT component loads fully linked SmartDeck applications directly onto the card without the need to generate an Application Load Unit (ALU). The files we load onto cards are regular SmartDeck executables and have the extension `.hzx`. These contain the code and static data of the application, together with the application's session data requirements and other details such as the application identifier.

What's in a certificate?

Because MULTOS emphasises security it requires **load certificates** for loading applications onto the card and **delete certificates** for deleting them from the card must be supplied.

The SDT component supports two ways of supplying MULTOS load and delete certificates:

- auto-generated certificates
- user-supplied certificates The SDT component can auto generate certificates during the load and delete operation for MULTOS development cards. The certificates that are generated match the application footprint precisely, but generate default values for the application- provider public key. If a specific application-provider public key is required or you have special requirements for the application footprint, then you'll need to get some certificates generated by an issuing authority. User-supplied certificates are held in files with the extension `.alc` for load certificates and `.adc` for delete certificates. The suitability of a certificate is checked against the application footprint prior to its use for program load.

Application characteristics

The main characteristics of an application (the code size, data size and session data size) can be deduced from the application itself. Other characteristics are supported using object file attributes. The attributes that are supported by the SDT component define:

- application identifier entry— the value of the attribute aid is interpreted as a string of hexadecimal numbers (which can be space seperated) for example `#pragma attribute("aid", "f0 00 10 00")` would result in the application having the application identifier `f0001000`.
- file control information entry — the value of the attribute `fci` is interpreted as a string of hexadecimal numbers.
- the directory file entry — the value of the attribute `dir` is interpreted as a string of hexadecimal numbers.
- whether the application is a shell application — the attribute `shell` defines if the application is a shell application.
- whether the application is a default application — the attribute `default` defines if the application is a default application.
- which of the ATR historical characters the application accesses — the

attribute `atrtype` value can take the value `"primary"` or `"secondary"`, the default value is no access.

The MULTOS directory file

When the SDT component loads an application, the MULTOS directory file is updated with an entry that records the AID and the corresponding application name. This enables the component and the user to identify the applications loaded on a card, and provides a friendly way of referring to applications (e.g., `eloyalty` rather than `F0000104`).

If an attribute has been used to specify a user supplied entry in the directory file then the SDT component will put this in rather than construct a directory entry. If this has been done then the application cannot be referenced by name instead the application identifier must be used for example

```
hterm -selectaid "f0 00 01 04" -apdu "70 30 00 00 02"
```

```
hterm -deleteaid "f0 00 01 04"
```

A directory entry that hasn't been put in by the SDT component will be displayed as a string of hexadecimal numbers.

Shell applications

When loading and deleting applications the SDT component accesses the MULTOS and manufacturer data of the card to determine the card type and accesses the MULTOS directory file. If you are developing a Shell application then you can either support the MULTOS commands

- select MF/DIR file
- read record,
- Get Manufacturer Data
- Get MULTOS Data in your Shell application (an example is provided in the distribution) or you will

have to use the SDT component in such a way so that it never needs to use the MULTOS commands.

The cardtype can be specified to the hterm program which will remove the use of GetManufacturerData and GetMULTOSData.

```
hterm -load shell.hzx -cardtype HitachiV4
```

```
hterm -deleteaid "f0 00 01 04" -cardtype HitachiV4
```

The use of the application identifer in the deletion avoids referencing the directory file.

Initialisation at load time

To assist development of applications that are to be personalised the SDT component provides a facility to redefine the initial values of global variables when the application is loaded. This facility uses the symbolic names of the variable to allow for the changes that occur during program development.

If the following variable is declared in an application

```
int points;
```

It's value can be set at load time as follows

```
hterm app.hzx -init points=0064
```

Which will initialise the variable points to be 100. Note that the application must be built with debugging (`-g`) enabled to be able to use this facility.

ALU loading

To load or delete an application load unit a suitable load/delete certificate must be supplied.

```
hterm -alu app.alu -alc confkeyv4.alch
```



```
term -adc confkeyv4.adc
```

The SDT component will check that the application footprint fits into the load certificate, but it won't check that the certficate is suitable for the card.

Command-line options

`hterm` is a client of the SmartDeck Terminal COM automation component. It can be used for application management and simple interactive testing. You can invoke `hterm` by using the following syntax:

```
hterm [ options ]
```

options is a command-line option. Options are case sensitive and cannot be abbreviated. Options can be put into indirect files and referenced on the command line using `@`**indirectfile** syntax.

Switch	Description
`-readers`	List names of PC/SC readers.
`-adc` **certificate**	Delete using given application delete certificate.
`-alc` **certificate**	Load using the given application load certificate.
`-alcDataSize`	When opening the application, allocate the amount of data specified in the ALC rather than in the .hzx file (Note, this swich is not supported with the –alu switch).
`-alu`	The file named in the -load switch is an ALU
`-apdu` **command**	Send APDU command.
`-atr`	Display ATR.
`-card`	Display type of card.
`-cardtype` **type**	Specify type of card; one of HitachiV3, HitachV4, KeycorpInfineonV4, KeycorpInfineonV4P MI-M3, MI-M4, MI-M5, MI-S4, ersa192, ersa256.
`-clean`	Delete all applications.
`-delete` **name**	Delete named application.
`-deleteaid` **aid**	Delete application identifier by certification generation.
`-dir`	Display MULTOS directory
`-init` **sym**=**val**	Redefine the initialisation of a global static variable. The value is a string of hexadecimal numbers. If the symbol is a pointer then the locations pointed at are updated.
`-interact`	Starts a command prompt. Allows multiple APDUs to be entered. A blank line exits the command prompt and the tool.
`-ip` **ipaddress**	Connect to the MULTOS device using a supported mobile phone contactless reader app available at the supplied ip address.
`-load` **file**	Load **file** onto the card (a .hzx file unless -alu is specified).
`-logfile` **name**	Log all APDU commands to named file. The filename stdout can be used to direct output to the console.
`-manufacturer`	Display manufacturer data

`-multos`	Display MULTOS data
`-name` **appname**	Give a name to the loaded application (default is to use the basename of the load file).
`-pcsc` **readerName**	Connect to named PC/SC reader (if omitted will connect to an arbitrary reader).
`-pcsc` **readerNum**	Connect to a numbered PC/SC reader (zero based index).
`-pcscdllname name`	Specify the name of the .dll through which hterm will access PCSC.
`-reload` **file**	Reload **file** onto the card, requires that the application name is the basename part of the filename.
`-select` **name**	Select named application .
`-selectaid` **aid**	Select application by application identifier .
`-serial` **COM#:bps**	Connect to MULTOS device using serial port. e.g. COM4:19200 (serial interface) or COM5:i2c (using http://www.robot-electronics.co.uk/htm/usb_iss_tech.htm module)
`-sim` **name**	Connect to simulator/debugger with the name given to the `-ifd` option of the simulator/debugger
`-V`	Display version
`-verbose`	Show information messages

Table 11 Card Tool options

`hterm` executes the options in the following order:

1. **turn on logging/verbose**
2. **connect to PC/SC reader or simulator**
3. **display options (atr, manufacturer, multos dir)**
4. **initialise global variables**
5. **set the card type, if specified**
6. **clean or delete application**
7. **delete the application if reload has been requested**
8. **load the application**
9. **select application**
10. **send APDU commands**
11. **disconnect from the PC/SC reader or simulator**

Using the SDT component

This section demonstrates and describes more-advanced use of the SDT component. `hterm` is a useful program to get applications loaded onto cards, but it can't be used to load and test an application automatically—for example, regression testing an application after a change. In such cases, you'll need to script your tests, and the SDT component will make life a lot easier for you.

Background information

A COM automation server is a dynamically linked component that can be called from a COM client (such as `hterm`) or from a scripting language such as JScript.

This section defines the syntax of the methods of the SDT component. Unless we say otherwise, all

examples are given in JScript (Microsoft's version of JavaScript) or ECMAScript.

Component creation

The programmatic identifier (ProgId) of the SDT component is "`SmartDeck.Terminal`". Before you can interact with the card, you must create a new instance of this component.

JScript example

```
term = new ActiveXObject("SmartDeck.Terminal")
```

VBScript example

```
Dim term Set term = CreateObject("SmartDeck.Terminal")
```

Connecting to a smart card reader

You can connect the SDT component to a PC/SC card reader by using the `connectpcsc` method, giving the name of the reader as a parameter. In this case, we're connecting to a Gemplus GPR400, or GemPC400.

Example

```
term.connectpcsc("Gemplus GPR400 0");
```

It's important to get the name of the reader right: spaces and case are important. If the name you give doesn't match a known reader, the component returns an error.

Connecting to the SmartDeck simulator

You can connect to a SmartDeck simulator or debugger using the `connectsim` method, passing the name of a communication object as a parameter.

Example

```
term.connectsim("simulator");
```

The name must match the name given to the debugger with the `-ifd` option. For example, you would start a debugger using:

```
hdb -ifd simulator eloyalty.hzx
```

This starts the debugger, loads the eloyalty application, and waits for input on the communication object named `simulator`. Or you can use a quiet console-mode simulator:

```
hsim -ifd simulator
```

Resetting a card

You can reset a card using the `reset` method. This method takes an integer parameter that specifies whether the card should be powered down (cold reset).

Example

```
term.reset(0); // warm resetterm.reset(1); // cold reset
```

Disconnecting a reader

The `disconnect` method will disconnect from the card reader or terminate the simulator/debugger.

Example

```
term.disconnect();
```

Redefining values of global static variables

To enable development of applications that are to be personalised the methods `setdata`, `setdatalen` and `setdatabyte` can be used. The `setdata` method takes the name of a global static variable and a string of hexadecimal numbers. When the application is loaded the data memory corresponding to the variable is overwritten with the supplied initialisation data. For example a C program could contain the following global variable


```
unsigned char pin[4];
```

by default this variable will be zero'd by the toolset. If the following is used

```
term.setdata("pin", "aa bb cc dd");
```

then it's initial value will be equivalent to the C program

```
unsigned char pin[4] = { 0xaa, 0xbb, 0xcc, 0xdd };
```

The SDT component doesn't do any type or size checking when it does the initialisation, so if your initialisation strings are too big they will initialise things they ought not to. The methods `setdatalen` and `setdatabyte` provide the same functionality as `setdata` but can handle binary data.

```
term.setdatalen("pin",4);term.setdatabyte("pin",0,0xaa);term.setdatabyte("pin",1,0xbb
);term.setdatabyte("pin",2,0xcc);term.setdatabyte("pin",3,0xdd);
```

Note that the application must be compiled with debugging (`-g`) enabled to be able to use this facility.

Loading & deleting applications by certificate generation

The method `loadbycertgen` will load a `.hzx` file by generating an application load certificate based on the application footprint and the type of card that is in the card reader.

```
term.loadbycertgen("app","d:\app.hzx");
```

…will load the file `d:\app.hzx` and create a suitable entry in the directory file.

The method `deletebycert`gen will delete the named application by generating a delete certificate which matches the application identifier of the application and detecting the type of card that is in the card reader.

```
term.deletebycertgen("app");
```

You can specify the card type if for example you have a shell application that doesn't support the get MULTOS command.

```
term.setcardtype("HitachV4");
```

The method `deleteaidbycert`gen will delete an application by generating an application delete certificate based on the supplied application identifier.

```
term.deleteaidbycertgen("f000ff01");
```

...will delete the application by constructing an application delete certificate for the application identifier `f000ff01`. This method of deletion should be used when the MULTOS directory file cannot be accessed or the application has it's own entry in the MULTOS directory file.

The method `cleanbycertgen` will delete all loaded applications by examining the MULTOS directory file and deleting each entry by generating a suitable delete certificate.

```
term.cleanbycertpath();
```

Loading & deleting applications by fixed certificate

The method `loadbycert` will load a .hzx file with a given application load certificate. This method will check that the application footprint fits the application load certificate.

```
term.loadbycert("app","d:\app.hzx","d:\app.alc");
```

…will load the file `d:\app.hzx` using the certificate `d:\app.alc` and create an entry in the directory file of the application identifier contained in the application load certificate and the name `app`. The method `deletebycert` will delete using an application delete certificate.

```
term.deletebycert("d:\app.adc");
```

The method `loadALU` can be used to load a MULTOS application load unit with a given application certificate

```
term.loadALU("d:\app.alu",”d:\app.alc”);
```


Application selection and sending APDU commands

The method `selectbyname` takes the name of an application, finds the corresponding application identifier (by consulting the MULTOS directory file), and selects that application.

```
term.selectbyname("app");
```

Alternatively the method `selectbyaid` selects the application using its application identifier.

```
term.selectbyaid("f000ff01");
```

The file control information returned by the select can be examined using the `getresponse*` methods (see below). The method `setcommand` takes a string containing two-byte, optionally space- delimited hex numbers and puts it into the APDU command buffer of the component.

```
term.setcommand("70 30 00 00 02");
```

The method `exchange` sends the contents of the APDU command buffer to the card or simulator.

```
term.exchange();
```

The method `getresponse` returns the last APDU command response as a two-byte, space-delimited string of hexadecimal numbers.

```
var response = term.getresponse();
```

Rather than working in strings, APDU commands can be manipulated as binary data with the `setcommandlen` and `setcommandbyte` methods. `setcommandlen` defines the number of bytes of the command; `setcommandbyte` sets a particular byte value in the APDU command buffer.

```
var cmd = new Array(0x70, 0x30, 0x00, 0x00, 0x02);
term.setcommandlen(cmd.length);
for (var i = 0; i < cmd.length; i++)
  { term.setcommandbyte(i, cmd[i]);}
```

Similarly, `getresponselen` and `getresponsebyte` can be used to extract the APDU command response as binary data.

```
var out = "";
for (var i = 0; i < term.getresponselen(); i++)
  out += term.getresponsebyte(i).toString(16) + " ";
```

Examining the state of the card

The ATR, MULTOS data, manufacturer data, and directory file can be seen via the methods `getatr`, `getmultosdata`, `getmanufacturerdata`, and `getdirectory`.

The `getatrlen` and `getatrbyte` methods can be used to return the ATR string of the card as binary data.

```
var out = "ATR " for (var i = 0; i <
term.getatrlen(); i++)out +=
term.getatrbyte(i).toString(16) + "
";WScript.echo(out);
```

The `getmultosdata` and `getmanufacturer` methods return strings containing the fields of the MULTOS and manufacturer data of the card, respectively.

```
WScript.echo("multosdata:")WScript.echo(term.getmultosdata());WScript.echo("manufactu
rer data:")WScript.echo(term.getmanufacturerdata());
```

The `getdirectory` method returns a string containing the contents of the MULTOS directory file. This string consists of a line for each entry, in the form:

```
application name : application id
```

Unless a user defined directory entry has been placed in which case this entry is output as a string of hexadecimal numbers. If no applications are loaded, the string "`no applications loaded`" will be returned.


```
WScript.echo("directory:")WScript.echo(term.getdirectory());
```

Grab-bag methods

The method `logcommands` takes a filename as a parameter and causes all interactions between the component and the card/simulator to be recorded to the named file.

```
term.logcommands("test1.txt");
```

The `verbose` method takes an integer parameter (1 = on, 0 = off) that controls the display of informational messages as methods are executed.

```
term.verbose(1);
```

The `listpcscreaders` method returns a string containing a newline seperated list of PC/SC reader names.

```
WScript.echo(term.listpcscreaders());
```

Complete examples of the SDT component

The following script (in the file `ex1.js`) will connect to a PC/SC card reader, clean the card of any previously loaded applications, load an application, and send an APDU command.

```
term = new ActiveXObject("SmartDeck.Terminal"
term.connectpcsc("SmartCard Reader RSCR RS-232 1");
term.cleanbycertgen();
term.loadbycertgen("eloyalty", "eloyalty.hzx");
term.selectbyname("eloyalty");term.logcommands("ex1.log");
term.setcommand("70 30 00 00 02");
term.exchange();
term.disconnect();
```

You can run this script using the Microsoft Windows Script Host as follows

```
C:\SmartDeck\Scripts>cscript ex1.js
```

and will produce a file called `ex1.log` with the command and its response.

The Assembler

The assember `has` is responsible for translating SmartDeck assembly code files into object files.

Command syntax

You invoke the assembler using the following syntax:

```
has [ option ] file
```

Files

file is an ASCII source file containing an assembly code modules whose syntax is described in the Assembler User Guide.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The assembler supports the following command line options:

Option	Description
`-D`***name*** `[=`***val***`]`	Define the symbol ***name*** and optionally set it to ***val***.
`-D`***name***	Define the symbol ***name*** and set its value to be -1.
`-g`	Produce debugging information
`-I`***dir***	Add the library search directory to ***dir***
`-J`***dir***	Add the library search directory to ***dir***
`-o` ***file***	Leave output in ***file***
`-U`***name***	Undefined ***name***
`-w`	Suppress warnings
`-V`	Display version

Table 12 Assembler options

The C Compiler

The C compiler `hcc` is responsible for translating SmartDeck C code files into object files.

Command syntax

You invoke the C compiler using the following syntax:

```
hcc [ option ] file
```

Files

file is an ASCII source file containing C source code.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The C compiler supports the following command line options:

Option	Description
`-D`***name*** `[=`***val***`]`	Define the preprocessor symbol to be ***val***.
`-D`***name***	Define the preprocessor symbol.
`-g`	Produce debugging information
`-I`***dir***	Add dir to the end of the user library search path.
`-J`***dir***	Add dir to the end of the system library search path.
`-no_locals_opt`	Do not optimise locals when using -opt*n* option.
`-o` ***file***	Leave output in ***file***
`-opt`***n***	Use optimised instruction group **n***
`-O`	Optimise output
`-Ods`	Disable using static when compiling switch statements
`-U`***name***	Undefined pre-processor symbol ***name***
`-c`	Allow C++ single line comments
`-V`	Display version
`-ansi`	Warn about non-ANSI C source code.

Table 13 C Compiler options

* Where n > 0 this currently optimises the layout of local variables on the stack to ensure that when linked the optimal instruction coding is used. In the event that you wish to prevent this behaviour for some functions you can use the *-no_locals_opt* option or the compiler pragmas **fixlocals** and **fixlocalsoff**. E.g.:

```
void f2(void)
{
	char s1[16];
	uint8_t x;
	char s2[32];
	uint16_t i;
```



```
        s1[0] = 0;
        s2[0] = 0;
        x = 1;
        i = 1;
}

// As f2() but within fixlocals, so no optimisation will happen
#pragma fixlocals
void f2_protected(void)
{
        char s1[16];
        uint8_t x;
        char s2[32];
        uint16_t i;

        s1[0] = 0;
        s2[0] = 0;
        x = 1;
        i = 1;
}
#pragma fixlocalsoff

void f3(void)
{
...
```


The Linker

The linker `hld` is responsible for linking together the object files which make up your application together with some run-time startup code and any support libraries.

Although the compiler driver usually invokes the linker for you, we fully describe how the linker can be used stand-alone. If you're maintaining your project with a make-like program, you may wish to use this information to invoke the linker directly rather than using the compiler driver.

Linker function and features

The linker performs the following functions:

- resolves references between object modules;
- extracts object modules from archives to resolve unsatisfied references;
- combines all fragments belonging to the same section into a contiguous region;
- removes all unreferenced code and data;
- runs an architecture-specific optimizer to improve the object code;
- fixes the size of span-dependent instructions;
- computes all relocatable values;
- produces a linked application which can be in a number of formats.

Command syntax

You invoke the linker using the following syntax:

`hld [ option | file ]...`

Files

file is an object file to include in the link and it must be in SmartDeck object format. You do not give library files on the command line in this way, you specify then using the `-l` and `-L` switches described below.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated.The linker supports the following command line options:

Option	Description
`-A`***name*** `[=`***val***`]`	Define the attribute ***name*** and optionally set it to ***val***.
`-E`***name***	Set program entry symbol to ***name***
`-F`***fmt***	Set linker output format to ***fmt***
`-g`	Propatate debugging information
`-L`***dir***	Set the library search directory to ***dir***
`-l`***x***	Search library ***x*** to resolve symbols
`-M`	Display linkage map on standard output
`-O`	Optimise output
`-o` ***file***	Leave output in ***file***
`-opt`***n***	Use optimised instruction group 'n'. See [MDRM] for details.

-nocen	Do not issue Compact Enable instruction. See [MDRM] for details.
-pure	Ensure only code, no data
-T***name=addr***	Place section ***name*** at ***addr***
-V	Display version

Table 15 Object file lister option summary

Linker output formats

The linker supports a number of industry-standard file formats and also the native SmartDeck file format used by the debugger. The compiler driver arranges for the linker to generate whatever is the most appropriate format for the tool set in use. For the MEL target, the default is to write the application as an unencrypted application load unit.

The -F***fmt*** switch sets the output format, and the following formats are supported:

Format switch	Format description
-Fsrec	Motorola S-record
-Fhex	Intel extended hex
-Ftek	Tektronix hex
-Flst	Hex dump
-Frte	Microsoft SCW run-time environment RTE and DAT format
-Fhzx	SmartDeck native, used by the debugger
-Falu	MULTOS ALU

Table 16 Supported output formats

Laying out memory

The linker need to know where to place all code and data which make up an application. You tell the linker where to place each section using the -T switch.

Example

```
hld app.hzo -T.text=200 -T.bss=4000 -T.data=8000
```

This sets the .text section to start at address 200_{16}, the .bss section from 4000_{16}, and the .data section from 8000_{16}.

Example

You can assign where your own sections are placed. Assume you need to place the following set of vectors at address $FFFA_{16}$ as this is where the processor expects to find them.

```
.SECT ".vectors" DW irqDW nmi DW reset
```

You use the -T switch as above to set the address: `hld -T.vectors=fffa`

Linkage maps

You can find where the linker has allocated your code and data in sections by asking for a linkage map using the -M option. The map is a listing of all public symbols with their addresses.

Linking in libraries

You can link in libraries using the -lx option. You need to specify the directory

where libraries are to be found, and you can do this with the `-L` option.

Example

Link the library libcrypto.hza and libio.hza found in \SmartDeck\lib: `hld -L\SmartDeck\lib -lcrypto -llibio`

The Archiver

This section describes the archiver, or librarian, which you can use to create object code libraries. Object code libraries are collections of object files which are consolidated into a single file, called an archive. The benefit of an archive is that you can pass it to the linker and the linker will search the archive to resolve symbols needed during a link.

By convention, archives have the extension `.hza`.

Automatic archiving

The compiler driver hcl can create archives for you automatically using the `-ar` option. You will find this more convenient than manipulating archives by hand and we recommend that you use the compiler driver to construct archives.

Command syntax

You invoke the archiver using the following syntax:

```
har [ option ] archive file...
```

Files

archive is the archive to operate on.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The options which are recognised by the archiver are:

Option	Description
`-c`	Create archive
`-t`	List members of archive
`-V`	Display version of archiver

Table 17 Archiver options

Creating an archive

To create an archive you simply use `har` with the name of the new archive and list any files which you wish it to contain. The archive will ne created, overwriting any archive which already exists with the same name.

Example

To create an archive called `cclib.hza` which initially contains the two object code files `ir.hzo` and `cg.hzo` you would use:

```
har -c cclib.hza ir.hzo cg.hzo
```

The archiver expands wildcard file names so you can use

```
har -c cclin.hza *.hzo
```


Listing the members of an archive

To show the members which comprise an archive, you use the `-v` switch. The member's names are listed together with their sizes. If you only give the archive name on the command line, the archiver lists all the members contained in the archive. However, you can list the attributes of specific members of the archive by specifying on the command line the names of the members you're interested in.

Example

To list all the members of the archive `cclib.hza` created above you'd use:

```
har -t cclib.hza
```

To list only the attributes of the member `ir.hzo` contained in the archive `cclib.hza` you'd use:

```
har -t cclib.hza ir.hzo
```


The Object File Lister

The object file lister `hls` is a general-purpose program which can display useful information held in unlinked object files and linked executables. It can:

- show disassembled code
- show section addresses and sizes
- show exported symbols with their types and values
- show imported symbols with their types

Typically it's used to show object code generated by the SmartDeck compilers, or to produce an intermixed listing of code and high-level source.

Command syntax

You invoke the object filer using the following syntax:

```
hls [ option ]... file
```

Files

file is the object file to list.

- **Options**

option is a command-line option. Options are case sensitive and cannot be abbreviated. The object file lister supports the following command line options:

Option	Description
`-A`	Show object file attributes
`-e`	Show entry points
`-pt`	Generate patch table for multiple hzx files.
`-p`	Show public symbols defined in the module
`-P`	Show link order priority
`-perso`	Output the names and locations of data items declared in the *melperso* section (or *static* section if no *melperso* section is defined).
`-s`	Show all symbols defined or used in the module
`-src`	Include the
`-S`**`path`**	Add **path** to the list of paths to search for source files.
`-t`	Display section start address, end address and size.
`-v`	Show all object bytes generated in the listed
`-x`	Show externals
`-V`	Display version

Table 18 Object file lister option summary

Listing section attributes

You can quickly see a summary of each section in an executable using the `-t` switch. This presents a summary for each section showing its start address, its end address, its size in

decimal and hexadecimal, and the section name.

Example

```
hls -t app.hzx
```

This lists the contents of all sections in the file `app.hzx`.

The RSA Key-Pair Generator

This section describes the `hkeygen` program that generates RSA key-pairs. The `hkeygen` program takes as input the required modulus size and an optional exponent value and generates an appropriate RSA key-pair. The program generates a file containing the private part of the key-pair and a file containing the public part of the key-pair. The private part of the key-pair is generated in its Chinese Remainder Theorem form. The output files produced by the program can optionally be produced in a form suitable for inclusion into a C program. The default behaviour produces binary files that can be used for certificate generation or as the application key.

Command syntax

You invoke the key-pair generator using the following syntax:

```
hkeygen [ option ]...
```

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The options `-modsize`, `-private` and `-public` must be supplied. All the options which are recognised by the key-pair generator are defined in the following table.

Option	Description
`-cfile`	Output files are generated in a form suitable for inclusion into a C program.
`-exponent` ***n***	Specify the exponent value to be 'n' which is an arbitrary sized decimal number.
`-modsize` ***n***	Specify the modulus size in bits. Note that the modulus size must be divisible by eight.
`-private` ***f***	Specify the name of the file to contain the private part of the keypair.
`-public` ***f***	Specify the name of the file to contain the private part of the keypair.
`-v`	Verbose mode
`-V`	Display version

Table 19 Key-pair generator options

Binary output files

The output file format for the private part of the key-pair is

- a single byte that specifies thelength of the modulus, in bytes
- the public modulus
- the exponent zero-padded to modulus size
- the private modulus

- the private modulus in CRT form (p, q, dp, dq, u) each element being modulus/2 bytes long

The output file format for the public part of the key-pair is

- a single byte that specifies the length of the modulus, in bytes
- the public modulus
- the exponent zero padded to modulus size
- zero padding of (modulus size * 3.5) to make the file the same size as the private part.

C format output files

The `<rsa.h>` library is used when the key-pair generator creates output files suitable for inclusion in a C program. The private part of the key-pair contains an initialised definition of a struct variable of type `RSAPrivateCRTKey`. The public part of the key-pair contains an initialised definition of a struct variable of type `RSAPublicKey`. Both of the output files define the macros `MODLEN` and `EXPLEN` so if a compilation unit includes both files it will need to `#undef` these macros prior to the second inclusion.

For example the key-pair generator could be run as follows...

```
hkeygen -modsize 1024 -public pub.h -private priv.h -cfile
```

...which will generate the files `pub.h` and `priv.h` containing the public and private parts of the key-pair respectively. These files can be used in an application as follows:

```
#include <multoscomms.h>
#include <string.h>
#include "pub.h"
#undef MODLEN
#undef EXPLEN
#include "priv.h"
unsigned char plaintext[MODLEN];
unsigned char ciphertext[MODLEN];
unsigned char recoveredtext[MODLEN];
void main(void
{
  int i;
  for (i = 0; i < MODLEN; ++i)
    plaintext[i] = i;
  RSAEncipher(&publicKey, plaintext, ciphertext);
  RSADecipher(&privateKey, ciphertext, recoveredtext);
  if (memcmp(plaintext, recoveredtext, MODLEN)==0)
   ExitSW(0x6400);

 ExitSW(0x9000);

}
```

This example tests that the enciphered data can be succesfully deciphered (a process that `hkeygen` has already done as part of its verification process).

ALU Generation

This section describes the `halugen` program that generates standard MULTOS application load units (ALUs). The `halugen` program takes as input a linked executable file (.hzx) and can produce unprotected, protected or confidential ALU's as output.

To produce a protected or confidential ALU the tool must be supplied with an application private key which can be created using the `hkeygen` program. The corresponding public key should be used in the creation of the load and delete certificates for the application.

To produce a protected or confidential ALU the tool can either use default key values for development cards or alternatively key values can be supplied as input to the tool. For protected or confidential ALU's the key used by MULTOS to compute/verify the hash digest of the application must be supplied. To produce a confidential ALU the public key certificate of the MULTOS card and it's associated transport key certifying key must be supplied.

For confidential ALU's the default behaviour is to use a random number as the key for triple DES enciphering of the entire application. The DES key can be specified on the command line to the tool, the size of which will determine if single or triple DES enciphering is used.

Command syntax

You invoke the ALU generator using the following syntax:

```
halugen [ option ] filename
```

Options

option is a command-line option. Options are case sensitive, cannot be abbreviated and are defined below.

Option	Description
`-ahashk` ***f***	Specify the name of the file containing the key for the MULTOS Asymmetric Hash. This file should be in the same format as that of a public key file as generated by `hkeygen`. This option is required for producing protected and confidential ALU's if the `-cardtype` option isn't supplied.
`-appk` ***f***	Specify the name of the file containing the application private key. The file should be in the same format as that of a private key file as generated by `hkeygen`. This option is required for producing protected and confidential ALU's.
`-cardtype` ***ct***	Specify type of card; one of HitachiV3, HitachV4, KeycorpInfineonV4, KeycorpInfineonV4P, MI-M3, MI-M4, MI-M5, ersa192, ersa256.

`-confidential`	Produce a confidential ALU. This option requires the option -appk and either -cardtype or -ahashk, -mcdpkc, -tkck options. If the option - DESkey is not supplied a 16 byte random number is used as the key value for Triple DES enciphering of the application.
`-dataonly`	When used with the –confidential switch, this switch indicates that only the data section will be encrypted.
`-autoPad`	Automatically pad v4 AUs correctly for confidential ALU generation.
`-DESkey` "***key***"	Specify the DESkey for the enciphering of confidential ALU's. Either 8 or 16 byte keys are supported. The ***key*** is interpreted as a sequence of space seperated hexadecimal numbers. If this option isn't supplied then a random 16 byte number is used as the key for enciphering the ALU.
`-init` ***sym***=***val***	Redfine the initialisation of a global static variable. The value is a string of hexadecimal numbers. If the symbol is a pointer then the locations pointed at are updated.
`-mcdpkc` ***f***	Specify the name of the file containing the public key certificate of the MULTOS card for which the confidential ALU is being generated. This option is required for producing a confidential ALU without the ***- cardtype*** option.
`-o` ***filename***	Specify the filename the ALU is to be written to. By default this will be the name of the `.hzx` file with a `.alu` suffix.
`-pad` ***N***	Pad the code and data sections of the ALU so they are a multiple of ***N*** bytes. For MULTOS V3 the code and data must be padded to 32 byte multiples.
`-protected`	Produce a protected ALU. This option requires the option -appk and either -cardtype or -ahashk options.
`-tkck` ***f***	Specify the name of the file containing the public key to decipher the MULTOS public key certificate. This file should be in the same format as that of a public key file as generated by `hkeygen`. This option is required for producing confidential ALU's when the option – `cardtype` isn't supplied.
`-V`	Display version

Table 20 ALU generator options

ALC / ADC Generation

This section describes the **melcertgen** program that generates Application Load Certificates and Application Delete Certificates for MULTOS Developer cards. It is useful for when you wish to test with confidential and protected ALUs. Otherwise for plaintext ALUs the other SmartDeck tools will generate default certificates on the fly.

The application's properties (sizes, application ID, FCI record etc) can be extracted from an **hzx** file or can be provided via command line options individually.

Command syntax

melcertgen *certName* [options]

certName is the root name of the certificates to be generated. The tool generates an ALU and an ADC.

Options

Options are case sensitive, cannot be abbreviated and are defined below. Options marked with # are only required if the –hzx file option is not specified.

Option	Description
-aid a	Hexademical Application ID of the application being certified
-cs n	Code size in bytes
-ds n	Data size in bytes
-ss n	Session data size in bytes
-dfs n	Directory file size in bytes
-fcis n	FCI record size in bytes
-alist x	Specify access_list value (in decimal) of x – default is 3. This will override the value of the access_list attribute (if provided)
-pkey file	Binary file containing the application provider public key to be certified.
-hzx file	Application to produce the certificates for.
-addstat n	Add n extra bytes of static above the number found in the .hzx file to the ALC's static data size.
-mcdno mcd_number	8 bytes Hexadecimal mcd_number to be set in the certificates. Default is all zeros.
-pad n	Pad data sizes in hzx file to n bytes. The value of n for MULTOS v4 is 8.
-confidential	Produce a confidential ALC.
-protected	Produce a protected ALC.
-cardtype ct	Specify type of card; one of HitachiV3, HitachV4, KeycorpInfineonV4, KeycorpInfineonV4P, MI-M3, MI-M4, MI-M5, MI-S4, ersa192, ersa256
-adf	Generate a JSON formatted application definition file for the KMA.

Table 20b – ALC/ADC Generator Options

Example ALU and ALC generation commands for Developer Cards

The following commands show how ALUs and ALCs can be created for developer cards.

Plaintext ALUs

halugen Eloyality.hzx -o Eloyality_plain.alu

melcertgen Eloyality_plain_v4 -hzx Eloyality.hzx -cardtype MI-M3

Protected ALUs

halugen -cardtype MI-M3 -protected -appk app_provider.priv Eloyality.hzx -o Eloyality_prot_v4.alu

melcertgen Eloyality_prot_v4 -hzx Eloyality.hzx -pkey app_provider.pub -protected -cardtype MI-M3

Confidential ALUs

halugen -cardtype MI-M3 -confidential -appk app_provider.priv -dataonly -autoPad Eloyality.hzx -o Eloyality_conf_v4.alu

melcertgen Eloyality_conf_v4 -hzx Eloyality.hzx -pkey app_provider.pub -pad 8 -confidential -cardtype MI-M3

ALU generation commands for Community Cards

For community cards (which use live keys) the individual keys have to be specified instead of using the –cardtype option. ALCs need to be obtained from the MULTOS KMA.

Protected ALUs

halugen -protected -ahashk hashmod_0247.pub -appk app_provider.priv Eloyality.hzx -o Eloyality_prot_v4.alu

Confidential ALUs

halugen -tkck tkck0203.key -mcdpkc 1506005D2A56.pkc -ahashk hashmod_0247.pub -confidential -dataonly -autoPad –appk app_provider.priv Eloyality.hzx -o Eloyality_conf_v4.alu

MULTOS File Dump

This section describes the `meldump` tool that lists the contents of various files used in MULTOS application development. The tool can list the following file formats

- Application Load Units (ALU) including disassembly of the code section of an ALU.
- Application Load Certificates (ALC) .
- Application Delete Certificates (ADC) .
- MULTOS Public Key Certificates (PKC).
- RSA Key file (such as that produced by `hkeygen`).
- An arbitrary binary file.

With certificate files for development cards the public key they contain can be deciphered and the certficates can be authenticated.

Command syntax

You invoke the `meldump` tool using the following syntax: `meldump [ options ] filename`

If no options are supplied then the file is dumped as a hexadecimal/ascii.

Options

options are command-line options. Options are case sensitive, cannot be abbreviated and are defined below.

Option	Description
`-alu`	Interpret file as an application load unit.
`-alc`	Interpret file as an application load certificate.
`-adc`	Interpret file as an application delete certificate.
`-cardtype` ***ct***	Specify type of card; one of HitachiV3, HitachV4, KeycorpInfineonV4, KeycorpInfineonV4P, MI-M3, MI-M4, MI-M5, ersa192, ersa256
`-cfile varname`	Output file as a C initialised variable `varname`.
`-codehash`	Include the SHA-1 of the code section when using -alu option.
`-d`	Disassemble the code section of an ALU.
`-decipher`	Decipher the public key part of the certificate (`-cardtype` option required).
`-key`	Interpret file as an RSA key as produced by `hkeygen`.
`-pkc`	Interpret file as a MULTOS card Public Key Certificate.
`-tkck` ***f***	Specify the name of the file containing the public key to decipher the MULTOS public key certificate. This file should be in the same format as that of a public key file as generated by `hkeygen`. This option is required for producing confidential ALU's when the option – `cardtype` isn't supplied.

`-V`	Display version

Table 21 ALU generator options

Example commands

To list an ALU

```
meldump -alu eloyalty.alu
```

The command

```
halugen -alc mycert.alc -cardtype HitachiV4 -decipher
```

will list the contents of the ALC including the public key.

The Hex Extractor

The hex extractor `hex` is used to prepare images in a number of formats to burn into EPROM or flash memory. Although the linker is capable of writing all the formats described here, it doesn't have the capability of splitting files for different bus or device sizes.

Command syntax

You invoke the hex extractor using the following syntax:

```
hex [ option ]... file
```

Files

file is the object file to convert.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The hex extractor supports the following command-line options:

Option	Description
`-F`***fmt***	Select output format ***fmt***
`-o` ***file***	Write output with ***file*** as a prefix
`-T`***name***	Extract section ***name*** from input file
`-V`	Display tool version information
`-W`***width***	Set bus width to ***width***

Table 22 Hex extractor option summary

Supported output formats

The following output file formats are supported:

```
-Fsrec  Motorola S-record
-Fhex   Intel extended hex
-Ftek   Tektronix hex
-Flst   Hex dump
-Fapp   APP format for codelets
```

Preparing images for download

When you prepare to download applications to a monitor held in ROM, you usually need the application in a single industry-standard format such as S-records or Intellec hex format. What you don't need to do is split high and low bytes, and you won't need to split across ROMs.

The extractor generates files in this format by default—all you need to provide is the format you need the file in.

Example

```
hex -Fhex app.hzx
```

This will generate a single Intellect-format file, `app.hzx.hex`, which contains all code and data in the application. The addresses in the output file are the physical addresses of where the code and

data are to be loaded.

Preparing images for ROM

Preparing ROM images is more involved than preparing images to download using a monitor. You may need to split images across multiple ROMs, for instance to separate high and low bytes for a 16-bit bus, or to split a large application across multiple ROMs because a single ROM doesn't have the capacity to hold the whole application. The extractor copes with both of these requirements.

Example

```
hex -B256 app.hzx -Fhex
```

This splits the application file `app.hzx` into multiple pieces, each of which is 256 kilobits. The `-B` option matches the part names of most 8-bit EPROMs, so the application is split into 256 kilobit sections, or 32 kilobyte chunks. `-B256` could equally well have been specified using `-K32`.

Example

```
hex -W2 app.hzx -Fhex
```

This splits the application file `app.hzx` into high and low bytes for a processor with a 16-bit bus. The bytes at even addresses are placed in one file, and the bytes at odd addresses in a second.

Example

```
hex -W2 -B512 app.hzx -Fhex
```

This splits the application file `app.hzx` into high and low bytes for a processor with a 16-bit bus and also splits each of these into ROMs which have a 512 kilobit capacity. The bytes at even addresses are placed in one set of files, and the bytes at odd addresses in a second set of files.

Extracting parts of applications

By default the extractor will generate output files which contain the whole application. In some cases you may need to extract only the contents of certain sections or fixed range of addresses.

Extracting sections

If you need only to extract only certain sections from a file, you can use the `-T` option to specify those to extract.

Example

```
hex -T.text -T.vectors app.hzx
```

This will generate a single file, `app.hzx.bin`, which contains only the sections `.text` and `.vectors` from the file `app.hzx`.

Codelets

Codelets are a way to place commonly-used code into ROM or EEPROM so a single copy of the code can be shared by many applications. Because ROM uses much less die area that EEPROM, it's also a good way of squeezing bigger programs into the small space of a smart card.

Codelets are similar to shared libraries that you'll find in modern operating systems, and share some of the problems too. In the following sections you'll be shown how codelets can provide solutions to some common problems such as:

- Moving code from your application in EEPROM to a ROM codelet to reclaim data space in EEPROM.
- Share code that is common between one or more applications by moving the shared code into a codelet.
- How to co-ordinate and develop applications using more than one codelet.

A guide to setting up Eclipse for Codelet development follows this chapter.

Your first codelet

Let's start off with the simplest scenario where you have two functions in your application, and you'd like to move one of them into a codelet. For now we will assume that this function doesn't use any global data (that includes MULTOS static, public, and session data) as this complicates matters somewhat.

We'll break the application into two source files:

- `func.c` contains the source code of the function we'll be putting into the codelet;
- `main.c` contains the code of our main application that makes calls to the codeletized function.

Here are the two source files; they don't do anything astoundingly useful, but serve admirably for demonstrations purposes:

Source code for "func.c"

```
// Count number of set bits in a word
unsigned countBitsSet(unsigned x)
{
        int n = 0;
        while (x){
                x &= x-1;
                ++n;
        }
        return n;
}
```

Source code for "main.c"


```
// Main application
#include <multoscomms.h>
// External function prototype
unsigned countBitsSet(unsigned);

void main(void)
{
  int n = countBitsSet(0x5555); // 8 bits are set, 8 bits are clear
}
```

You know how to compile and link these together without using a codelet; you do it like this:

```
hcl main.c func.c
```

This links all object code into the main application and leaves the output in `main.hzx`. What we need to do now is move `func` out of the main application and into a codelet.

We do this in four steps:

1. Mark the functions you want to export as codelet entry points

You need to mark the functions you want exported as codelet entry points by writing "__codelet" after the function declaration in the module that defines the function. Using our example:

```
unsigned countBitsSet(unsigned x) __codelet // codelet entrypoint
{
  int n = 0;

  while (x){
   x &= x-1;
   ++n;
  }

  return n;

}
```

You need to mark the entry points because codelet functions use a calling convention that is slightly different from the regular calling convention.

2. Choose a codelet ID by which the codelet is known

All codelets that are included in a MULTOS card need distinct codelet IDs. You can request your own unique codelet ID from MAOSCO, and let's say that in this instance codelet ID 21 has been assigned to us for our example codelet.

3. Compile and link the codelet

To generate the codelet is simply a matter of using the `-codelet` option on the command line and supplying the codelet ID information:

```
hcl -codelet -Acodeletid=21 func.c
```

The tools write the file "`func.hzx`" that contains the code section of the codelet, and the file "`func.hzl`" that contains the information used by clients on resources the codelet needs, the codelet ID to use, what the exported functions can be called and how to call them.

4. Compile and link the client that uses the codelet

To use the functions exported by a codelet, all that clients need to do is link in the codelet library generated as part of the codelet linkage above—it's just like an additional object

module:

```
hcl func.hzl main.c
```

The tools take care of selecting the correct calling sequence to use and the special client-codelet interworking. Note that we didn't make any changes to the ***client*** program at all—the function prototype of `countBitsSet` used in `main.c` does ***not*** need the `__codelet` keyword to identify that it comes from a codelet. This makes it straightforward for you to move functions around in your codelets whilst they're in development without needing wholesale, error-prone modification of client code.

There is one thing to say, however: if you change the implementation of the codelet, all clients that use that codelet must be recompiled using the new codelet library (the .hzl file). This is necessary because of the nature of MULTOS and the way that codelets work.

Note:
How you get a codelet onto a MULTOS card, whether in ROM or EEPROM, is not covered here, and you will need to work closely with the MULTOS operating system implementer to deploy your codelet. However, as codelets are nothing more than pure MEL code it's worth knowing you can extract the code section of a linked file using the hex utility. This utility supports output in a number of formats including straight binary, Motorola S-record and Intel Hex. For example:

```
Hex –Fbin func.hzx –T.text
```

will extract the code as a straight binary image to `func.hzx.bin`.

Codelets that need static data

The whole point of putting code ROM using a codelet is to reduce the amount of EEPROM used on a MULTOS card. And, as such, it's a good bet that you have a lot of code and a large data requirement, so here we address the issue of how to go about using data in a codelet.

Codelets are pure code and have no data of their own

Codelets do not have static data allocated for them—they are pure code. Any global, static data that a codelet needs to refer to must be allocated in the application that calls the codelet.

Because codelets are pure code there are two methods that you can use to refer to static data within them:

- Allocating codelet static data in the application, with a structure known to the application and the codelet. Both the codelet and the application can address the static data directly using the MULTOS SB register.
- Pass the codelet a pointer to some of the application's static data; when the codelet needs to use static data, it can dereference the pointer passed to it.

Next, we'll examine both methods by presenting a single example in the two coding styles. This example performs a DES encoding in CBC mode using keys that are held in static and updated after each DES operation.

Allocating codelet static data in the application

The simplest way for a codelet to have static data is to let the compiler/linker allocate the codelet static data in the client application. This approach allows the source code of a

codelet to be written like conventional C code and is efficiently implemented by the compiler/linker. Unfortunately using this approach an application cannot link with more than one codelet that requires
static data (note however that several pure code codelets can also be linked in).

Passing static data by reference

This section will examine how a codelet can reference static data through a pointer passed to it. This mechanism is well known and widely used in many architectures, usually to allow shared code.

First we'll present a function that will be placed into a codelet that does some DES-based cryptography where the plaintext and ciphertext addresses are passed to it, but the DES keys are held in static data as they don't change.

We'll define a "global" structure that holds the information for use by the codelet; in this case we hold some DES keys and the information used to derive those keys. We'll place this structure into a header file along with the function declarations that will work with this data.

Code for "session.h"

```
#ifndef SESSION_H
#define SESSION_H
typedef struct
{
	// Static encryption key
	unsigned char staticKey[8];
	// Authentication encryption key
	unsigned char authEncKey[8];
	// Key derivation data
	unsigned char derivationData[8];
	// And anything else we might wish to keep hold of...
} GlobalData;

void genDerivationData(GlobalData *globals,unsigned char *hostChallenge,unsigned char
*cardChallenge);

void genAuthEncKeys(GlobalData *globals);
void verifyCryptogram(GlobalData *globals,unsigned char *cryptogram);
#endif
```

Now we can write the code that implements these functions:

Code for "session.c"

```
#include "session.h"
void genDerivationData(GlobalData *globals,unsigned char *hostChallenge,unsigned char
*cardChallenge) __codelet
{
	memcpy(globals->derivationData, hostChallenge, 2);
```



```
        memcpy(globals->derivationData+2, cardChallenge, 2);
        memcpy(globals->derivationData+4, hostChallenge+2, 2);
        memcpy(globals->derivationData+6, cardChallenge+2, 2);
}

void genAuthEncKeys(GlobalData *globals) __codelet {
        // Encrypt derivation data with static keys giving session keys
        DESEncipher(globals->derivationData,globals->authEncKey,globals->staticKey);
}
void verifyCryptogram(GlobalData *globals) __codelet
{
        // More wacky crypto stuff here
}
```

Notice that all static data is referenced through the global data structure. Now we'll write a code for the client application; the client needs to allocate the static data used by the codelet functions and pass the address of that data to each codelet function when it calls it.

Code for "main.c"

```
        // Main application

        #include <multoscomms.h>
        #include "session.h"

                    // Static storage used by the codelet.static GlobalData globals ={
                     // Static key{ 0xAA, 0xAA, 0xAA,
                     0xAA, 0xAA, 0xAA, 0xAA, 0xAA }};

                    // Static storage for us.static unsigned char hostChallenge[8];static
                    unsigned char cardChallenge[8];void main(void){
                      switch (INS){case INITIALIZE:
                          // Store host
                          challenge.memcpy(hostChallenge, apdu,
                          8);// Generate a random card
                          challenge.GenerateRandom(cardChallenge)
                          ;

                          // Stir the key derivation data.genDerivationData(&globals,
                          hostChallenge, cardChallenge);

                          // And generate the authentication encryption session
                          keys.genAuthEncKeys(&globals);

                          // Send card challenge back to
                          host.memcpy(apdu, cardChallenge,
                          8);ExitLa(8);

                        case EXTERNAL_AUTHENTICATE: // Verify
                          the host
                          cryptogram.verifyCryptogram(&globals
                          , apdu);

                          // If we get here the cryptogram was verified...
```

This example is a little contrived, but explains the idea.

Caveats

Disabling the code generator's use of static data

The SmartDeck code generator uses various heuristics and optimization techniques to reduce the code and data size of applications. However, some of these optimizations make use of tables in static storage, for instance `switch` statements can effectively use jump tables held in static.

If you're compiling code for a codelet, the last thing you want is the compiler allocating jump tables into static storage, so there is a compiler switch to disable the use of static storage when compiling.

This switch, `-Ods`, forces the compiler to use different code generation techniques to ensure that it doesn't require static data.

If you compile your source files using the `-codelet` switch, the `-Ods` switch is automatically sent to the compiler.

Avoiding static data in codelets

To ensure that a codelet doesn't require static data the `-pure` option can be used on the command line to `hcl`. Note that static data will be required if your codelet contains any literal strings.

Function pointers and codelets

Functions in codelets are called by a special mechanism that differs significantly from a regular call. The compiler and linker work together to make calling codelet functions just like calling regular functions ***at the source code level***.

At the machine level there is a difference, and the compiler and linker cannot cope with calling functions in codelets through a function pointer. If you call a codelet function through a function pointer, the stack layout will not be set up as the codelet expects and will lead to unpredictable behaviour.

At present neither the compiler nor linker will diagnose taking the address of a codelet or calling a codelet through a function pointer, so beware.

How to customise codelets and patch up broken code

This section describes how to write a codelet so it can be updated after the codelet has been placed into ROM. Obviously, if a bug is found in a codelet and needs to be fixed, and the codelet has been committed to ROM and can't be patched, a new mask is required: having a capability to replace a defective function in ROM with a correct one in EEPROM is vital.

Tool support for patchable codelets

The C compiler recognises the qualifier "`__patchable`" after a function declaration. This informs the tools that enough scaffolding should be put in place so that, if required, a function's implementation can be replaced.

Consider our previous codelet:

```
#include "session.h"

void genDerivationData(GlobalData *globals,unsigned char *hostChallenge,unsigned char
                       *cardChallenge) __codelet
{
  memcpy(globals->derivationData, hostChallenge, 2);
  memcpy(globals->derivationData+2, cardChallenge, 2);
  memcpy(globals->derivationData+4, hostChallenge+2, 2);
  memcpy(globals->derivationData+6, cardChallenge+2, 2);
}
```



```
  void genAuthEncKeys(GlobalData *globals) __codelet {
    // Encrypt derivation data with static keys giving session keys
    DESEncipher(globals->derivationData,globals->authEncKey,globals->staticKey);
  }
void verifyCryptogram(GlobalData *globals) __codelet
{
      // More wacky crypto stuff here
}
```

Each of these functions is non-patchable, so the functionality of each is fixed. In order to allow these functions to be patched, you would use the `__patchable` qualifier rather than the `__codelet` qualifier, so the code now becomes:

```
  #include "session.h"

  void genDerivationData(GlobalData *globals,unsigned char *hostChallenge,unsigned char
                         *cardChallenge) __patchable
  {
    memcpy(globals->derivationData, hostChallenge, 2);
    memcpy(globals->derivationData+2, cardChallenge, 2);
    memcpy(globals->derivationData+4, hostChallenge+2, 2);
    memcpy(globals->derivationData+6, cardChallenge+2, 2);
  }

  void genAuthEncKeys(GlobalData *globals) __patchable
  {
    // Encrypt derivation data with static keys giving session keys
    DESEncipher(globals->derivationData,globals->authEncKey,globals->staticKey);
  }

  void verifyCryptogram(GlobalData *globals) __patchable{
    // More wacky crypto stuff here}
```

The tools now arrange to call each of these functions using a table held in the host application's static area in EEPROM. The table in EEPROM initially points to the codelet entry points in ROM. However, because the table is held in the application, not in the codelet, the table can be updated to point to replacement functions provided by the application. Should we wish to replace a function, for instance `verifyCryptogram`, then we simply override it's implementation in our application program by declaring this function with the "`__patched`" qualifier like this:

```
  // Main application

  #include <multoscomms.h>

  #include "session.h" ...

  // Replace codelet version of verifyCryptogram
  void verifyCryptogram(GlobalData *globals) __patched
  {
    // Corrected wacky crypto stuff here
  }

  void main(void
  {
     switch (INS) ...
```

Notice that the function is declared `__patched` so that the compiler generates the correct function prologue and epilogure for a codeletised function. It's that simple! The tools take

care of constructing the patch table so building the patch table is something that you just don't need to consider. Now, all function calls to `verifyCryptogram`, whether from the host application or by the codelet, will call the replacement function in the application.

Help I've lost my .hzl file

When you build a codelet there are two files produced, the .hzx file containing the codelet and the .hzl file that describes the codelet to client applications. The .hzx file (actually the instructions hex extracted from it) will hopefully find it's way into the ROM of the MULTOS operating system. At some point in time a new application may wish to use this codelet and at this point the .hzl file needs to be used. Hopefully you have safely put said .hzl file into your code repository so there shouldn't be a problem. If for some reason you can't find the original .hzx file (or you have built a codelet via different technology) then (assuming you know details of the codelet) the `hzlgen` tool can be used to generate a new .hzl file from a textual description of the codelet.

The input to the hzlgen tool is a text file that describes each of the codelet entry points and optionally a description of the static data requirements of the codelet. Consider the following example codelet description

```
codeletid 12

function _apatchableCodeletEntrypoint 0 patchable

function _nonPatchableCodeletEntrypoint 12

function _cmain 49

data _sharedCodeletStaticData .SB 0x04 2 00 0e

data _privateCodeletStaticData .SB 0x06 2

data L10 .SB 0x8 12 70,6f,73,74,3a,63,6d,61,69,6e,0a,00
```

A line that starts with `codeletid` defines the codelet identifier of the codelet, the syntax is.

```
codeletid <cid>
```

Where `<cid>` is a C style number (0x for hex, 0 for octal and defaults to decimal). Lines that start with `function` declare function entry points into the codelet, the syntax of function entry point declarations is

```
function <functionname> <offset> [patchable]
```

Where `<functionname>` is the name of the function, `<offset>` is the byte offset to the function start (specified as a C style number) and `patchable` is a modifier if the function is patchable.

Lines that start with `data` define the static data requirements of the codelet and have the following syntax

```
data <symbolname> <sectionname> <offset> <size> [initialiser]
```

Where `<symbolname>` is the name of the variable, `<sectionname>` is the name of the section the variable is allocated in (either .SB, .DB or .PB), `<offset>` is the byte offset from the start of the section (specified as a C style number), `<size>` is the size of the variable in bytes (specified as a C style number). An optional initialisation sequence can be supplied as a comma seperate list of hexadecimal numbers.

The hzlgen tool takes an input file and by default produces a .hzl file with the same filename. The -o option can be used to rename the output .hzl file.

The overhead of codelet calls

In this section we deal with the overhead imposed by codelet functions.

Static overhead of a codelet function

Each function that is marked `__codelet`, `__patchable` or `__patched` uses a function prologue and epilogue sequence that differs from non-codelet functions. Depending upon the

formal parameter types and the function return type, this can be up to an extra four bytes of code per function that takes care of codelet linkage — this is per function, and not per function call.

Each function that is marked `__patchable` requires four bytes of static memory for an entry in the patch table held in the host application. If a function is marked `__codelet`, there is no overhead as a patch table entry is not required.

Dynamic overhead of a codelet function

All calls to functions marked `__codelet` or `__patchable`, from within the same codelet, from another codelet, or from an application, require an additional two bytes of stack to save the codelet ID (CI) register at the point of the codelet call.

Static overhead of calling a codelet function

Calls to functions ***within the same linked unit*** (either codelet or application) that are not marked `__codelet` nor `__patchable` use a three-byte instruction at each call site. In this instance this means a function call within a codelet to a function in the same codelet, or a function in an application to another function in the same application.

Calls to functions marked `__codelet` from an application or from another codelet require an eight-byte call sequence at each call site. A call to a function marked `__codelet` ***from the same codelet*** requires a six byte call sequence at each call site.

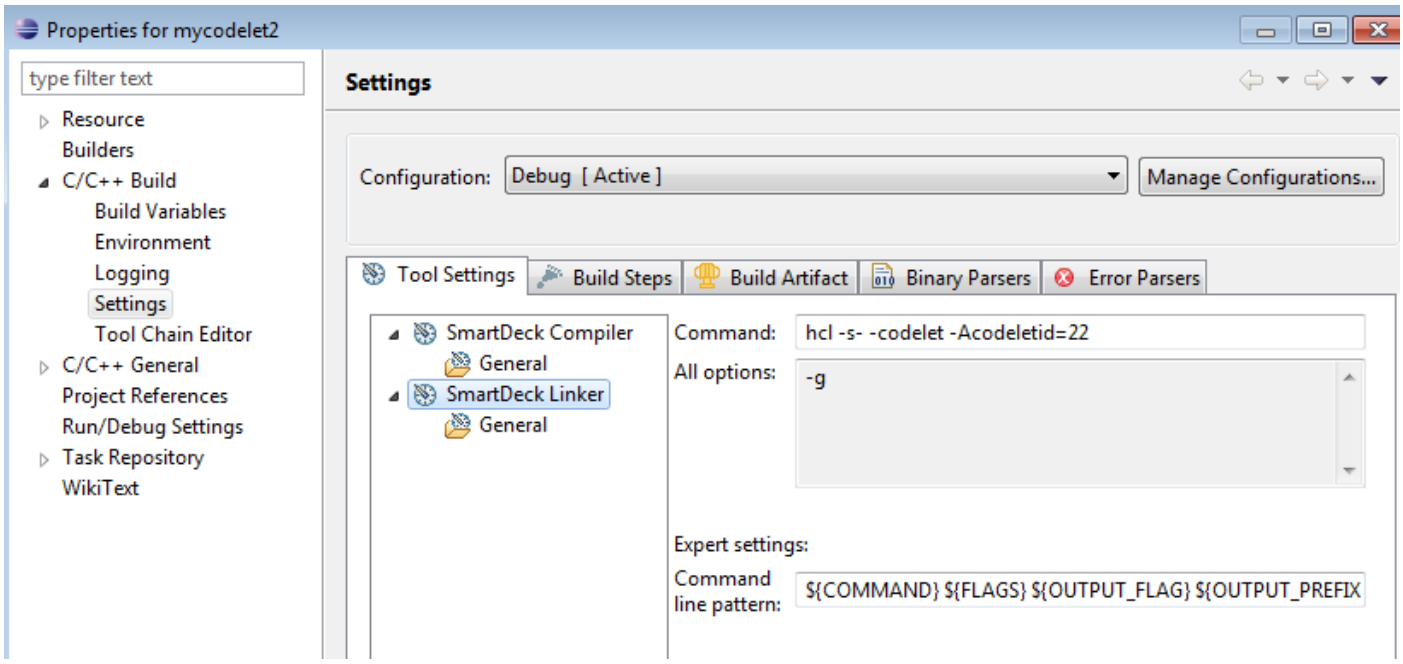
A call to a function marked `__patchable` requires an eight-byte call sequence at each call site irrespective of where the call site is located (a call through the patch table is always generated).

Setting up Eclipse for Codelet Development

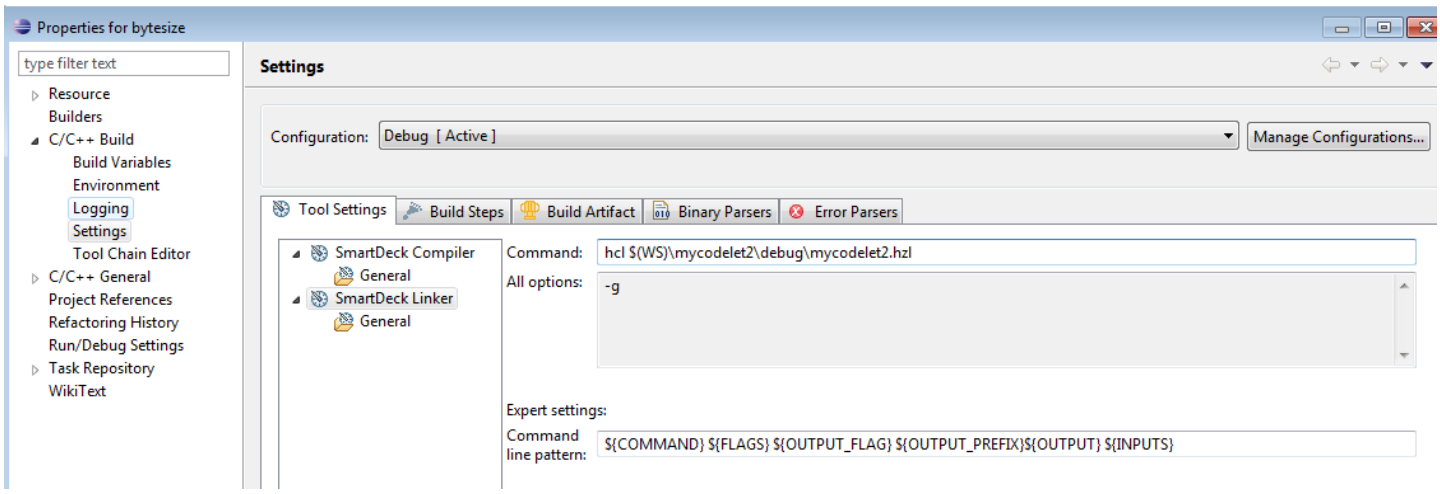
Basic Codelets

Linker Settings

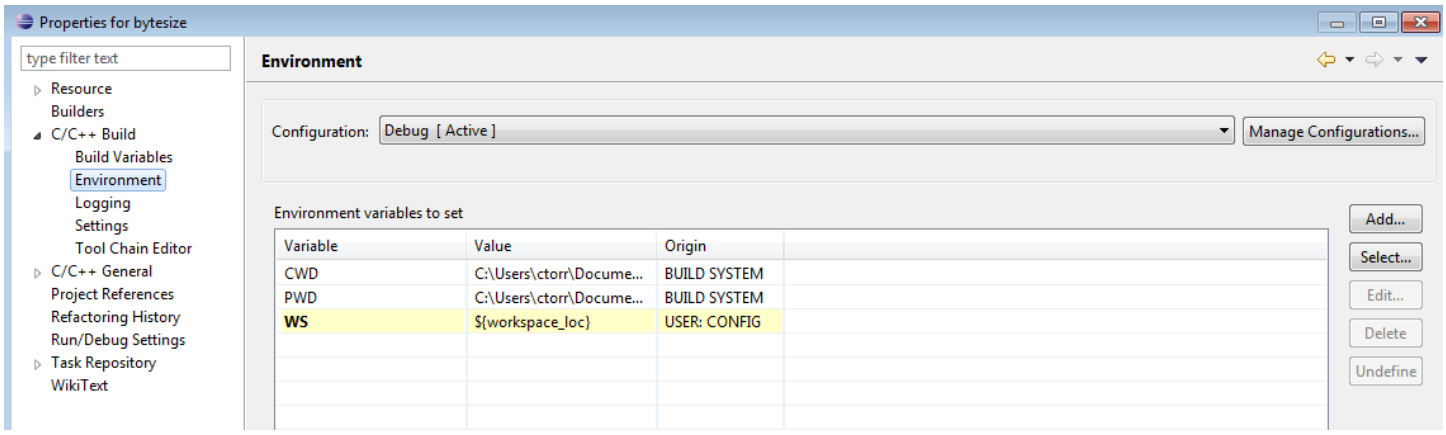
To build a project as a codelet you need to change the linker settings for the codelet project as follows: the SmartDeck user guide explains the switches. However, please read Advanced Codelet Development if you intend to start using multiple patchable codelets.



To build the "main" application so that it uses the codelet you need to change its linker settings too by telling it the library file to use:



$WS is an environment variable set in the Environment section - you could use anything.



IMPORTANT NOTES:

1. You can set up dependencies on codelet projects under **Properties->C/C++ General->Paths and Symbols->References.**
2. If you have source code files with the same name in your main app and codelet, the debugger will get very confused, especially when setting breakpoints.
3. You can have more than one codelet
4. Codelets calling other codelets is supported.

Debugging Codelets

In order to debug using codelets, you need to add the full path to the HZX files in the debug.txt (or whatever you have called it) file. e.g.

```
c:\program files (x86)\smartdeck\bin\hsim_dbg.exe
f3000003
-selectaid f3000003
-apdu 7001000002
C:\Users\ctorr\Documents\eclipse-multos\mycodelet2\Debug\mycodelet2.hzx
```

Advanced Codelet Development

The following instructions apply if

a. Your application makes use of more than one codelet with patchable functions OR
b. Two or more codelets are interdependent (i.e. they both call functions within each other)

The Issues Involved

It is important to remember that codelets **are not** the same as code libraries. A codelet is a fully linked, executable piece of code, whereas a library contains unlinked fragments of code. Also, once built and masked into the product at manufacturing time, codelets cannot be modified.

In order, therefore, for a function in a codelet to be patched, it is necessary to put new code in the application. Codelet functions trying to call the new function can only do so if the call is made indirectly by refering to a patch (jump) table held in the application's static memory (remember the codelet code can't be changed as it is held in immutable memory).

At the time the codelet is built it needs to know what offset in the patch table to use for each function. This is fine if you have one codelet with patchable functions. However, when you have two or more such codelets, independently generated patch tables for each codelet are going to clash. In order to provide a unique set of offsets for a cluster of related codelets it is therefore necessary to pre-allocate the offsets and pass this list to each codelet build and the final application link. In SmartDeck, this external list is a simple text file and is passed to the link command using the **-pt switch**. More on this later.

The other issue comes when you have two codelets that call each other. Because codelets are fully linked you cannot link one without the linker file (hzl) of the other, which can't be created because that codelet cannot be linked either for the same reason. i.e. there is a circular dependency. To break that dependency, the functions must be declared as *__patchable* and the external patch table used. Then then both rely on the external patch table to achieve the link.

The Patch Table File

This is a text file structured as follows, one line per codelet entry point.

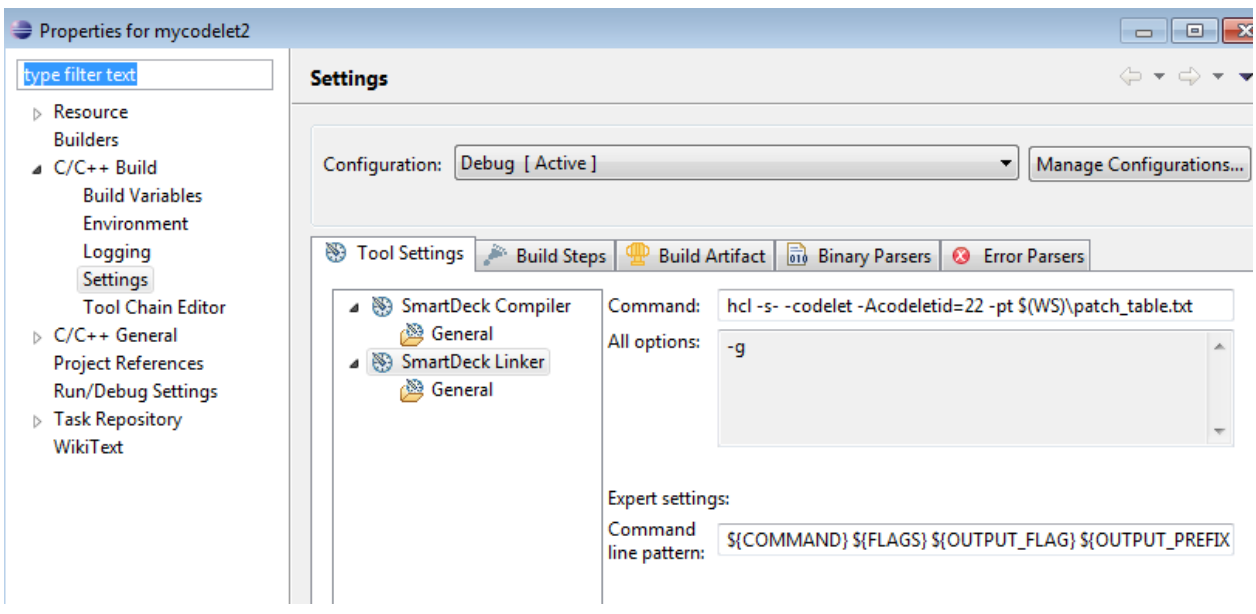
<Symbol_name> <codeletId - decimal> <patch table index>

For example

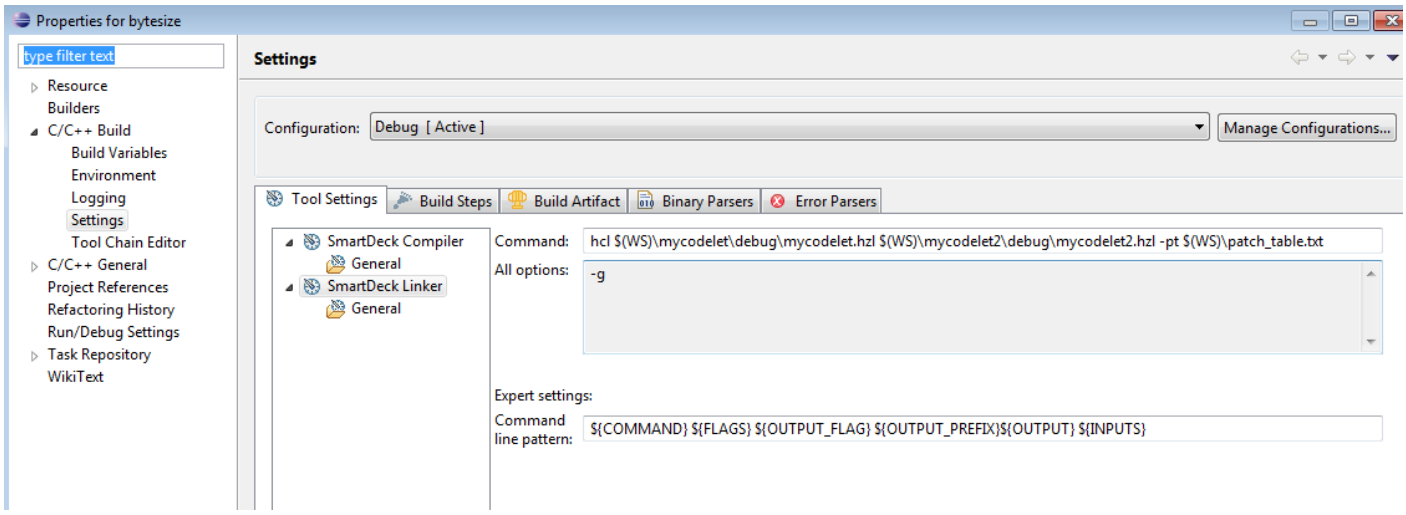
```
_codeletA_function1   22   0
_codeletA_function2   22   1
_codeletB_function1   23   2
_codeletC_function1   24   3
```

Each column is separated by white space (spaces or tab characters). The **<patch table index>** must be unique. Gaps in the sequence are allowed - the linker will automatically create a blank entry in the application's patch table.

The patch table is passed to the linker for each codelet as shown in this example:



The final application link also requires the patch table switch, as well as the linker (hzl) files for each codelet.



Building the Patch Table File

You can do this manually. Alternatively the SmartDeck application **hls** has been modified such that it can be used to extract the list of patchable entry points for a codelet. This could be useful if there are too many to type accurately by hand into the patch table file.

To use **hls** for this, do a preliminary build of the codelets without using the **-pt** switch. Then, in one go, run **hls** on all the generated **hzx** files as shown in the example below…

hls -Fapp codelet1.hzx codelet2.hzx codelet3.hzx > patch_table.txt

At this stage, you can manually modify the table as necessary - for example if you are making use of existing codelets that can no longer be modified.

Finally relink the codelets (after a "clean") with the -**pt** switch.

Assembler User Guide

The SmartDeck assembler converts assembly source code to relocatable object code written to object code files. The linker takes these object code files and combines them to form an ***Application Load Unit*** (ALU) containing the final instructions for use on the smart card.

Life is never simple, and software production for smart cards has complexities just like life itself. This introduction will skip a few details, both minor and major, which you'll need to know about when trying out your smart card applications—thankfully, we can put these to one side for now, which is probably rather unlike real life.

About this manual

The conventions used in the manual are described in the "Installation and Getting Started" section.

What we don't tell you...

This manual is a reference for the SmartDeck assembler. What we don't do in this manual is explain the architecture of the MULTOS virtual machine or how to go about constructing a smart card application.

The format of an assembly statement

Assembly language files are constructed from assembly language mnemonics and directives, collectively known as **source statements**. An assembly source module is a sequence of such source statements.

A statement is a combination of mnemonics, operands, and comments the defines the object code to be created at assembly time. Each line of source code contains a single statement.

Assembler statements take the form:

Syntax

[label] [operation] [operands] [comment]

Field	Purpose
label	Labels the statement so that the statement can be accessed by name in other statements
operation	Defines the action of the statement
operands	Defines the data to be operated on by the statement
comment	Describes the statement without affecting assembly

Table 23 Fields used in assembler statements

All fields are optional, although the operand or label fields may be required if certain directives or instructions are used in the operation field.

Label field

The label field starts at the left of the line, with no preceding spaces. A label name is a sequence of alphanumeric characters, starting with a letter. You can also use the dollar sign '$' and underline character '_' in label names.

A colon may be placed directly after the label, or it can be omitted. If a colon is placed after a label,

it defines that label to be the value of the location counter in the current section.

Operation field

The operation field contains either a machine instruction or an assembler directive. You must write these in either all upper-case or all lower-case— mixed case is not allowed.

The operation field must not start at the left of the line; at least one space must precede it if there is no label field. At least one space must separate the label field and the operation field.

Operand field

The contents of the operand depend upon the instruction or directive in the operation field. Different instructions and directives have different operand field formats. Please refer to the specific section for details of the operand field.

Comment field

The comment field is optional, and contains information that is not essential to the assembler, but is useful for documentation. The comment field must be separated from the previous fields by at least one space.

Constants

You can use constants to specify numbers or strings that are set at assembly time.

Integer constants

Integer constants represent integer values and can be represented in binary, octal, decimal, or hexadecimal. You can specify the radix for the integer constant by adding a radix specified as a suffix to the number. If no radix specifier is given the constant is decimal.

Syntax

decimal-digit digit... [B| O | Q | D | H]

Radix	Specifier
Binary	B
Octal	O or Q
Decimal	D
Hexadecimal	H

Table 24 Integer constant radix suffixes

Radix suffixes can be given either in lower-case or purchase letters. Hexadecimal constants must always start with a decimal digit (0 to 9). You must do this otherwise the assembler will mistake the constant for a symbol —for example, `0FCH` is interpreted as a hexadecimal constant but `FCH` is interpreted as a symbol.

Examples

```
224 ; 224
224D ; 224
17Q ; 15
100H ; 256
```

You can specify hexadecimal constants in two other formats which are popular with many assemblers:

Syntax

```
0x digit digit...
$ digit digit...
```

The `0x` notation is exactly the same as the way hexadecimal constants are written in C, and the `$` notation is common in many assemblers for Motorola parts.

Example

```
0xC0 ; 192
$F ; 15
```

String constants

A string constant consists of one or more ASCII characters enclosed in single or double quotation marks.

Syntax

```
'character...'
"character..."
```

You can specify non-printable characters in string constants using escape sequences. An escape sequence is introduced by the backslash character '\'. The following escape sequences are supported:

Escape Sequence	Description
ooo	Octal code of character where **o** is an octal digit
\"	Double quotation mark
\\	Backslash
\'	Single quotation mark
\b	Backspace, ASCII code 8
\f	Form feed, ASCII code 12
\n	New line (line feed), ASCII code 10
\r	Carriage return, ASCII code 13
\t	Tab, ASCII code 9
\v	Vertical tab, ASCII code 11
\x**hh**	Hexadecimal code of character where **h** is a hexadecimal digit

Table 25 Escape sequence in strings

Examples

```
"This is a string constant"
'A string constant with a new line at the end\n'
```

Comments

To help others better understand some particularly tricky piece of code, you can insert comments into the source program. Comments are simply informational attachments and have no significance for the assembler.

Comments come in two forms: single-line comments and multi-line comments.

Single-line comments

A single line comment is introduced either by single character `;` or by the characters //.

Syntax

```
// character...
; character...
```

The assembler ignores all characters from the comment introducer to the end of the line. This type of comment is particularly good when you want to comment a single assembler line.

Example

```
LOADI dataPtr, 1 // fetch next byte from APDU
INCN dataPtr ; increment data pointer
```

Which style you choose is down to personal preference.

Multi-line comments

A multi-line comment resembles a standard C comment as it is introduced by the characters `/*` and is terminated by `*/`.

Syntax

```
/* character... */
```

Anything in between these delimiters is ignored by the assembler. You can use this type of comment to place large amounts of commentary, such as copyright notices or functional descriptions, into your code.

Example

```
/* Elliptic curvecryptography library
Copyright (c) 1999, 2000 Dyne-O-Mite Software,Inc.
*/
```

It's quite acceptable to use this form to comment a single line, but using the single- line comment form is stylistically better.

Include files

You can include the contents of another file in your application using the `INCLUDE` directive.

Syntax

```
INCLUDE "filename"
INCLUDE <filename>
```

If the name of the file is contained in quotation marks, the assembler searches for the file on the user include path set by the `-I` command line option; if it's not found on this path, the assembler continues by searching the system include path (set by the `- J` option).

If the name of the file is contained in angle brackes, the assembler searches only the system include path set by the `-J` option.

Registers

The AAM register set is composed of seven address registers and two control registers,

shown in Table 26

Register Name	Register Contents
SB	Static base, the base of the static data area
ST	Static top, the extent of the static data area
DB	Dynamic base, the base of the dynamic data area
DT	Dynamic top, the extent of the dynamic data area
PB	Public base, the base of the public data area
PT	Public top, the extent of the public data area
LB	Local base, the base of the active stack frame

Table 26 Register names and uses

Addressing Modes

MEL addresses are broken into two classes:

- tagged addresses, and
- segmented addresses.

A tagged address is formed by specifying a constant offset to one of the internal AAM registers at the assembler level and the assembler only deals with tagged addresses. The AAM converts the tagged address to a segmented address at run-time when it adds an offset to the register and forms a 16-bit segmented address.

For assembler purposes, tagged addresses are further divided into two classes:

- explicit tagged addresses, and
- implicit tagged addresses.

All tagged addresses must specify a register as a base, and yet doing so is cumbersome when programming because you need to remember which section a data item was defined in.

In order to circumvent this problem, you can omit the register from the address and leave it for the **linker** to insert the correct register into the tagged address depending upon which segment the data item was defined in. This makes it much less of a burden to write assembler code because the linker takes care of the housekeeping. You can, of course, still specify a register when constructing a tagged address and the assembler and linker will honour your choice of register, even if it is the incorrect one.

Explicit tagged addresses

An explicit tagged address is written using the format

Syntax

register [offset]

For example, `SB[20]` is a tagged address as is `DT[-4]`. Any register can be used to construct a tagged address.

Implicit tagged addresses

An implicit tagged address is written without using a register. The register is filled in by the linker when the symbol is defined.

Labels, Variables and Sections

This section explains how to define labels, variables, and other symbols that refer to data locations in sections. The assembler keeps an independent ***location counter*** for each section in your application. When you define data or code in a section, the location counter for that section is adjusted, and this adjustment does not affect the location counters of other sections. When you define a label in a section, the current value of the location counter for that section is assigned to the symbol. You'll be shown how to assign labels and define most types of variables. You'll also be introduced to directives that control the location counter directly.

Label names

A label name is a sequence of alphanumeric characters, starting with a letter. You can also use the dollar sign '$' and underline character '_' in label names.

Symbolic constants or equates

You can define a symbolic name for a constant using the EQU directive. The EQU directive declares a symbol whose value is defined by an expression.

Syntax

symbol EQU ***expression***
symbol = ***expression***

The assembler evaluates the expression and assigns that value to the symbol.

Example

```
CR EQU 13
```

This defines the symbol CR to be a symbolic name for the value 13. Now you can use the symbol CR in expressions rather than the constant. The expression need not be constant or even known at assembly time; it can be any value and may include complex operations involving external symbols.

Example

```
ADDRHI EQU (ADDR>>8) & 0FFH
```

This defines the symbol ADDRHI to be equivalent to the value of ADDR shifted right 8 bits and then bitwise-anded with 255. If ADDR is an external symbol defined in another module, then the expression involving ADDR cannot be resolved at assembly time as the value of ADDR isn't known to the assembler. The value of ADDR is only known when linking and the linker will resolve any expression the assembler can't.

Using type specifiers

Some statements need data type specifiers to define the size and type of an operand. Data type specifiers are fully described in " Data Types and Expressions" , but for the moment the following will suffice:

Type name	Size in bytes	Description
BYTE	1	Unsigned 8-bit byte
WORD	2	Unsigned 16-bit word
LONG	4	Unsigned 32-bit word
CHAR	1	8-bit character
ADDR	2	16-bit address

Table 27 Assembler built-in types

Labels

You use labels to give symbolic names to addresses of instructions or data. The most common form of label is a ***code label*** where code labels as operands of call, branch, and jump instructions to transfer program control to a new instruction. Another common form of label is a ***data label*** which labels a data storage area.

Syntax

label [**:**] [***directive*** | ***instruction***]

The label field starts at the left of the line, with no preceding spaces. The colon after the label is optional, but if present the assembler immediately defines the label as a code or data label. Some directives, such as `EQU`, require that you do not place a colon after the label.

Example

```
ExitPt: RET
```

This defines `ExitPt` as a code label which labels the `RET` instruction. You can branch to the `RET` instruction by using the label `ExitPt` in an instruction: `JMP ExitPt`

Defining and initialising data

You can allocate and initialise memory for data using data definition directives. There are a wide range of data definition directives covering a wide range of uses, and many of these have the same semantics.

Defining simple data

You can initialise data items which have simple types using a convenient set of directives.

Syntax

label [**:**] [`DB` | `DW` | `DL` | `DD`] ***initialiser*** [**,** ***initialiser***]...

The size and type of the variable is determined by the directive. The directives used to define an object in this way are:

Directive	Meaning
`DB`	Define bytes
`DW`	Define words (2 bytes)
`DD DL`	Define long or double word (4 bytes)

Table 28 Simple data allocation directives

If the directive is labeled, the label is assigned the location counter of the current section before the data are placed in that section. The label's data type is set to be an array of the given type, the bounds of which are set by the number of elements defined.

Example

```
Power10         DW      1, 10, 100, 1000, 10000
```

This defines the label `POWER10` and allocates five initialised words with the given values. The type of `POWER10` is set to `WORD[5]`, an array of five words, as five values are listed.

Defining string data

You can define string data using the `DB` directive. When the assembler sees a string, it

expands the string into a series of bytes and places those into the current section.

Example

```
BufOvfl      DB     13, 10, "WARNING: buffer overflow", 0
```

This emits the bytes 13 and 10 into the current section, followed by the ASCII bytes comprising the string, and finally a trailing zero byte.

Aligning data

Data alignment is critical in many instances. Defining exactly how your data are arranged in memory is usually a requirement of interfacing with the outside world using devices or communication areas shared between the application and operating system.

You can align the current section's location counter with the `ALIGN` directive. This directive takes a single operand which must be a type name.

Syntax

`ALIGN` ***type***

The data type given after the directive defines the alignment requirement; if this type has a size ***n***, the location counter is adjusted to be divisible by ***n*** with no remainder.

Example

```
ALIGN LONG
```

This aligns the location counter so that it lies on a 4-byte boundary as the type `LONG` has size 4. To align data to an ***n***-byte boundary you can use the following:

```
ALIGN BYTE[n]
```

Filling areas

In many cases you'll need to fill large areas of code or data areas with zeroes or some other value. The assembler provides the `.SPACE` and `.FILL` directives for this purpose.

Filling with zeroes

A particular example of this is reserving memory space for a large array. Rather than using multiple data definition directives, you can use the `.SPACE` directive.

Syntax

[***label***] `.SPACE` ***n***

The `.SPACE` directive generates ***n*** bytes of zeroes into the current section and adjusts the location counter accordingly.

Example

```
.SPACE 10
```

This reserves 10 bytes in the current section and sets them all to zero.

Filling with a particular value

You may find it convenient to use the `.FILL` directive to fill an area with a certain number of pre-defined bytes. `.FILL` takes two parameters, the number of bytes to generate and the byte to fill with.

Syntax

[***label***] `.FILL` ***size***, ***value***

The `.FILL` directive generates ***size*** bytes of ***value*** into the current section and adjusts

the location counter accordingly.

Example

```
.FILL 5, ' '
```

This generates five spaces into the current section.

Using sections

You can use sections to logically separate pieces of code or data. For instance, many processors need interrupt and reset vectors placed at specific addresses; using sections you can place data at these addresses. Another use of sections is to separate volatile data from non-volatile data in an embedded system with dynamic RAM and battery-backed static RAM.

You select a section using the `.SECT` directive. If the section name doesn't exist, a new section is created.

Syntax

```
.SECT "section-name"
```

By convention, section names start with a period character. The section name is not an assembler symbol and can't be used in any expression. The assembler places all code and data into the selected section until another section is selected using `.SECT`.

Example

```
          .SECT ".vectors"
irqvec DW irq
nmivec DW nmi
rstvec DW reset
```

This example shows you how to set up a section named `.vectors` and populate it with some address.

Pre-defined sections

The SmartDeck system uses a number of pre-defined sections for MULTOS data:

Section name	Description
`.text`	Code section
`.PB`	MULTOS public section
`.DB`	MULTOS dynamic section
`.SB`	MULTOS static section

Table 29 Pre-defined sections

Data Types and Expressions

Unlike many assemblers, the SmartDeck assembler fully understands data types. The most well-known and widely used assembler that uses data typing extensively is Microsoft's MASM—and its many clones. So, if you've used MASM before you should be pretty well at home with the concept of data types in an assembler and also with the SmartDeck implementation of data typing.

If you haven't used MASM before you may well wonder why data typing should ever be put into an assembler, given that many assembly programs are written without the help of data types at all. But there ***are*** many good reasons to do so, even without the precedent set by Microsoft, and the two most valuable benefits are:

- The ability to catch potential or real errors at assembly time rather than letting them through the assembler to go undetected until applications are deployed in the field.
- Data typing is an additional and effective source of program documentation, describing the way data are grouped and represented.

We don't expect you to fully appreciate the usefulness of assembly-level data typing until you've used it in an application and had first-hand experience of both benefits above. Of course, it's still possible to write (almost) typeless assembly code using the SmartDeck assembler if you should wish to do so, but effective use of data typing is a real programmer aid when writing code.

Lastly, we should just mention one other important benefit that data typing brings and that is the interaction between properly-typed assembly code and the debugger. If you correctly type your data, the debugger will present the values held in memory using a format based on the type of the object rather than as a string of hexadecimal bytes. Having source-level debugging information displayed in a human-readable format is surely a way to improve productivity.

Built-in types

The SmartDeck assembler provides a number of built-in or pre-defined data types. These data types correspond to those you'd find in a high-level language such as C.

Type name	Size in bytes	Description
`BYTE`	1	Unsigned 8-bit byte
`WORD`	2	Unsigned 16-bit word
`LONG`	4	Unsigned 32-bit word
`CHAR`	1	8-bit character
`ADDR`	2	16-bit address

Table 30 Assembler built-in types

You can use these to allocate data storage; for instance the following allocates one word of data for the symbol `count`:

```
count VAR BYTE
```

The directive `DF` allocates one byte of space for `count` in the current section and sets `count`'s type to `BYTE`.

Structure types

Using the `STRUC` and `FIELD` directives you can define data items which are grouped

together. Such a group is called a ***structure*** and can be thought of as a structure in C.

Structured types are bracketed between `STRUC` and `ENDSTRUC` and should contain only `FIELD` directives.

Example

From an unshameful British stance, we could declare a structure type called `Amount` which has two members, `Pounds` and `Pence` like this:

```
Amount      STRUC
Pounds      FIELD       LONG
Pence       FIELD       BYTE
            ENDSTRUC
```

The field `Pounds` is declared to be of type `LONG` and `Pennies` is of type `BYTE` (we can count lots of Pounds, and a small amount of loose change).

Structured allocation and field access

The most useful thing about structures, though, is that they act like any built-in data type, so you can allocate space for variables of structure type:

```
Balance VAR Amount
```

Here we've declared enough storage for the variable `Balance` to hold an `Amount`, and the assembler also knows that `Balance` is of type `Amount`. Because the assembler knows what type `Balance` is and how big an `Amount` is, you can load the contents of `Balance` onto the stack in a single instruction:

```
LOAD Balance
```

This loads three bytes onto the stack (an amount is one word plus one byte). What's more, because the assembler tracks type information, you can specify which members of balance to operate on:

```
LOAD Balance.Pence
```

Here, we load a single byte onto the stack which is the `Pence` part of `Balance`. We could equally well have written:

```
LOAD Balance.Pounds
```

which loads the `Pounds` part of `Balance`.

Nested structures

Because user-defined structures are no different from the built-in types, you can declare fields within other structures to be of structure type. Taking the example above a little further, you could define a type `Account` to have two members, `Balance` and `ODLimit` which correspond to an current account's balance and overdraft limit:

```
Account     STRUC
Balance     FIELD       Amount
ODLimit     FIELD       Amount
            ENDSTRUC

MyAccount   DF          Account
```

Having a variable `MyAccount` declared of type `Account`, you can access the `Pounds` field of `MyAccount`'s `ODLimit` member using the dotted field notation:

```
LOAD MyAccount.ODLimit.Pounds
```

Array types

You can declare arrays of any predefined or user-defined type. Arrays are used extensively in high-level languages, and therefore we decided they should be available in the SmartDeck assembler to make integration with C easier.

An array type is constructed by specifying the number of array elements in brackets

after the data type.

Syntax

type `[` ***array-size*** `]`

This declares an array of ***array-size*** elements each of data type ***type***. The ***array-size*** must be an absolute constant known at assembly time.

Example

The type

```
BYTE[8]
```

declares an array of eight bytes.

Pointer types

You can declare pointers to types just like you can in most high-level languages.

Syntax

type `PTR`

This declares a pointer to the data type ***type***.

Example

The type

```
BYTE PTR
```

declares a pointer to a byte. The built-in type `ADDR` is identical to the type `BYTE PTR`.

Combining data types

Arrays, combined with structures, can make complex data structuring simple:

```
BankAccount   STRUC
HolderName    FIELD  CHAR[32]
HolderAddr    FIELD  CHAR[32]
Balance       FIELD  Amount
ODLimit       FIELD  Amount
              ENDSTRUC
```

Accessing array elements

You can select individual elements from an array by specifying the index to be used in brackets:

```
LOAD MyAccount.HolderName[0]
```

The assembler defines arrays as zero-based, so the fragment above loads the first character of `MyAccount`'s `HolderName`. Because the assembler must know the address to load at assembly time, the expression within the square brackets must evaluate to a constant at assembly time. For example, the following is invalid because `Index` isn't an assembly-time constant:

```
Index  DF     BYTE
       LOAD   MyAccount.HolderName[Index]
```

However, if `Index` were defined symbolically, the assembler can compute the correct address to encode in the instruction at assembly time:

```
Index EQU 20 LOAD MyAccount.HolderName[Index]
```

Byte and word extraction operators

These operators allow you to extract bytes from values at assembly time. They are **HIGH, HBYTE, LOW, LBYTE, HWORD and LWORD.**

Examples

```
HIGH   $FEDCBA98 ; evaluates to $BA
LOW    $FEDCBA98 ; evaluates to $98
HWORD  $FEDCBA98 ; evaluates to $FEDC
LWORD  $FEDCBA98 ; evaluates to $BA98
```

These can be combined to extract other bytes of values:

```
HBYTE HWORD $FEDCBA98 ; evaluates to $FE
LBYTE HWORD $FEDCBA98 ; evaluates to $DC
```

Index operator

The index operator indicates addition with a scale factor. It is similar to the addition operator.

Syntax

expression$_1$ [***expression***$_2$]

expression$_1$ can be any expression which has array type. ***expression***$_2$ must be a constant expression. The assembler multiplies ***expression***$_2$ by the size of the array element type and adds it to ***expression***$_1$.

Example

```
ARR VAR WORD[4] ; an array of four words
W3 EQU ARR[3] ; set W4 to the address ARR+3*(SIZE WORD)    ; which is ARR+6
```

Bit-wise operators

Bit-wise operators perform logical operations on each bit of an expression. Don't confuse these operators with processor instructions having the same names—these operators are used on expressions at assembly time or link time, not at run time.

Operator	Syntax	Description
NOT	NOT ***expression***	Bit-wise complement
~	~ ***expression***	
AND	***expression***$_1$ AND ***expression***$_2$	Bit-wise and
&	***expression***$_1$ & ***expression***$_2$	
OR	***expression***$_1$ OR ***expression***$_2$	Bit-wise inclusive or
|	***expression***$_1$ | ***expression***$_2$	
XOR	***expression***$_1$ XOR ***expression***$_2$	Bit-wise exclusive or
^	***expression***$_1$ ^ ***expression***$_2$	

Table 32 Logical operators

Examples

```
NOT 0FH             ; evaluates to FFFFFFF0
0AAH AND 0F0H       ; evaluates to A0
0AAH OR 0F0H        ; evaluates to FA
0AAH XOR 0FFH       ; evaluates to 55
```


Arithmetic operators

Arithmetic operators combine two expressions and return a value depending upon the operation performed.

Syntax

expression1 ***[operator]*** *expression2*

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
SHL or <<	Shift *expression1* left by *expression2* bits
SHR or >>	Logically shift *expression1* right by *expression2* bits
ASHR	Arithmentically shift *expression1* right by *expression2* bits

Table 33 Arithmetic operators

Examples

```
1 + 2          ; evaluates to 3
-1 SHR 3       ; evaluates to 0x1FFFFFFF
-1 ASHR 3      ; evaluates to -1
```

Relational operators

Relational operators compare two expressions and return a true value if the condition specified by the operator is satisfied. The relational operators use the value one (1) to indicate that the condition is true and zero to indicate that it is false.

Syntax

expression1 ***[operator]*** *expression2*

The following table shows the relational operators and their meanings.

Operator	Description
EQ or ==	True is expressions are equal
NE or !=	True if expressions are not equal
LT or <	True if *expression1* is less than *expression2*
LE or <=	True if *expression1* is less than or equal to *expression2*
GT or >	True if *expression1* is greater than *expression2*
GE or >=	True if *expression1* is greater than or equal to *expression2*

Table 34 Relational operators

Relational operators treat their operands as signed 32-bit numbers, so `0FFFFFFFFH GT 1` is false but `0FFFF GT 1` is true.

Examples

```
1 EQ 2   ; evaluates to false (0)
1 NE 2   ; evaluates to true (1)
1 LT 2   ; evaluates to true
1 GT 2   ; evaluates to false
1 LE 2   ; evaluates to true
```



```
1 GE 2   ; evaluates to false
```

Logical operators

Logical (Boolean) operators operate on expressions to deliver logical results.

Syntax

The following table shows the syntax of the logical operators and their meanings.

Operator	Syntax	Description
`AND or &&`	***expression***$_1$ `AND` ***expression***$_2$	True if both ***expression***$_1$ and ***expression***$_2$ are true
`OR or ||`	***expression***$_1$ `OR` ***expression***$_2$	True if either ***expression***$_1$ or ***expression***$_2$ are true
`NOT or !`	`NOT` ***expression***	True if ***expression*** is false, and false if ***expression*** is true

Table 35 Logical operators

Examples

```
1 AND 0 ; evaluates to false (0)
1 && 1 ; evaluates to true  (1)
1 OR 2 ; evaluates to true
NOT 1 ; evaluates to false
```

Miscellaneous operators

Retype operator

The retype operator `::` allows you to override the data type of an operand, providing the operand with a new type.

Syntax

expression `::` ***type***

The expression is evaluated and given the type, replacing whatever type (if any) the expression had.

Example

```
wordvar DW 2

        LOAD wordvar::BYTE
```

In this example, `wordvar` has the type `WORD` because it is defined using `DW`. The load, however, loads only a single byte because `wordvar` is retyped as a `BYTE`. Because retyping does not alter the value of the expression, it only alters its type, the load will read from the lowest address of `wordvar`.

THIS operator

You can refer to the current value of the location counter without using a label using the `THIS` operator.

Syntax

```
THIS
```

The `THIS` operator returns an expression which denotes the location counter ***at the start of the assembler line***. This is very important: the location counter returned by `THIS` does not change even if code is emitted.

Example

A typical use of `THIS` is to compute the size of a string or block of memory.

```
MyString    DB     "Why would you count the number of characters"
```



```
                    DB      "in a string when the assembler can do it?"

      MyStringLen EQU THIS-MyString
```

DEFINED operator

You can use the `DEFINED` operator to see whether a symbol is defined or not. Typically, this is used with conditional directives to control whether a portion of a file is assembled or not.

Syntax

`DEFINED` ***symbol***

The `DEFINED` operator returns a Boolean result which is true of the symbol is define ***at that point in the file***, and false otherwise. Note that this operator only inquires whether the symbol is known to the assembler, not whether it has a known value: imported symbols are considered as defined even though the assembler does not know their value. `DEFINED` cannot be used to detect whether a macro has been defined.

Example

The following show how defined works in a number of cases.

```
     .IMPORT X
Y EQU 10
B1 EQU DEFINED X ; true (1)
B2 EQU DEFINED Y ; true (1)
B3 EQU DEFINED Z ; false (0) -- not defined yet
B4 EQU DEFINED U ; false (0) -- never defined
Z EQU 100
```

SIZEOF operator

You can use the `SIZEOF` operator to retrieve the size of a variable or expression, if that variable or expression has a type.

Syntax

`SIZEOF` ***expression***

The `SIZEOF` operator returns an integer value which is the size of the type associated with the expression. The assembler reports an error if the expression has no type.

Example

```
X       STATIC WORD[100]
XSIZE   EQU    SIZEOF X ; 200, 100 two-byte elements
X0SIZE EQU     SIZEOF X[0] ; 2, size of WORD
```


Block Structure

The SmartDeck assembler provides you with features to help with constructing programs using something akin to the functions and block structure of a high-level language. For instance, you can create assembler functions which take parameters and return results, just like C functions. The aim of this section is to guide you through the block structure features of the assembler so that you can apply them to your own programs.

You shouldn't have any problems understanding this section if you have used the MULTOS virtual machine before. If you haven't, we suggest that you become familiar with the MULTOS virtual machine first, and then tackle this section.

Two types of block

The SmartDeck assembler supports two block types, one roughly corresponding to a high-level function and the other to a block nested within another block or function. Not surprisingly, we'll refer to a block corresponding to a high-level function as a ***function block***, and a block nestling inside another block as a ***nested block***.

There are differences between the two types of block; the most notable differences are the way that the blocks are closed and that only function blocks can be given input and output parameters.

Function blocks

A function block is bracketed using `BLOCK` ... `RETURN` directives. The statements between these two directives constitute the body of the block and are treated as a unit. You'll appreciate the importance of function blocks when we come to construct applications from multiple object files or library routines.

Function blocks always end with a `RETURN` directive. Note that this is a ***directive*** and not an ***instruction***: the assembler will automatically generate one of the four variants of MEL `RET` instruction for you depending upon the contents of the block body. Because function blocks always end with `RETURN`, you should always call them using a `CALL` instruction.

Simple function blocks

To define a function block you bracket your statements with the `BLOCK` and `RETURN` directives. The `BLOCK` directive introduces the block, and the `RETURN` directive ends the block and generates an appropriate `RET` instruction. For instance, say we wish to define a function `ValAPDU` to validate the INS and CLA bytes of an APDU. `ValAPDU` should exit with an appropriate status if the bytes are not set correctly, and return to the caller for further processing if they are. We can construct the block like this:

```
ValAPDU     BLOCK
            CMPB        CLA, ISO_CLASS
            BEQ         ClaOK
            EXITSW      0x6E00
ClaOK       CMPB        INS, READ_RECORD
            BEQ         InsOK
            EXITSW      0x6A81
InsOK       RETURN
```

We can ignore the instructions in the function body for the moment and concentrate on the fact that the block has been defined with a label `ValAPDU`. Now, this function can be called from an appropriate place in our mainline code using the `CALL` instruction:

```
Main CALL ValAPDU
```

At the label `Main` the function `ValAPDU` is called, `ValAPDU` validates the contents of the INS and CLA bytes, and if all is well control is returned to the instruction immediately after the call.

Function blocks with input parameters

The function `ValAPDU` does a useful job—unfortunately it's rather rigid and we can't

use it to validate the INS and CLA bytes against anything other than the values ISO_CLASS and READ_RECORD. As such, ValAPDU is a specialised function; what would be more useful is if we could adapt it so that it's generalised rather than specialised.

Well, there's good news. Rather like C functions, assembler-level functions declared using BLOCK can take parameters. We can make the ValAPDU function more general if we pass to it the values to check the INS and CLA bytes against.

The mechanics of passing the INS and CLA bytes to check against is simple: push them onto the stack. So, we'd use a boilerplate code such as:

```
Main      PUSHB    ISO_CLASS     ; class to verify against
          PUSHB    READ_RECORD   ; instruction to verify against
          CALL     ValAPDU       ; do verification
```

With that out the way all we need to worry about is accessing the parameters passed to the function from within the function body. Here's the code fragment to do that:

```
ValAPDU    BLOCK
class      IN     BYTE
instr      IN     BYTE
           LOAD   class
           CMPN   CLA
           POPB
           BEQ    ClaOK
           EXITSW 0x6E00
ClaOK      LOAD insn
           CMPN INS
           POPB
           BEQ InsOK
           EXITSW 0x6A81
InsOK     RETURN
```

Comparing this with the specialised version above, you'll see that there are two new directives at the head of the block body and the way the comparisons are coded has changed.

The IN directive names the parameters passed into the function and declares their type. In this function there are two parameters, class and instr, both of type BYTE.

In the function body the parameters class and instr are loaded onto the stack just like any other declared data item. The assembler provides the correct addressing mode for the LOAD instruction and takes care of calculating the offset to be used. We check the class and instruction as before and always exit through the RETURN directive.

The RETURN directive generates the appropriate MEL RET instruction to drop the two parameters from the stack when the function returns. So, immediately following the call to ValAPDU, the two parameters that were pushed onto the stack have been consumed and the stack adjusted. As you can see, the function is now slightly longer and less compact than the original, but it is more flexible.

Parameter types and declaration order

The order in which you declare parameters matters! You must declare the parameter you push onto the stack first as the first IN parameter, the second as the second IN parameter, and so on. In the example above we pushed the expected class first, then the instruction; and in the function body we matched this order by declaring the parameter class first and then instr.

Not only that, you must make sure that the data that you push at the call site matches the data types of the parameters. Above, we pushed two bytes onto the stack before the call and this is matched with two byte parameter declarations in the function block.

Functions that return values

Up to now you've seen how to call a function both with and without parameters. There is another avenue to

explore and that's how to return values from function blocks.

Blocks, scope, and labels

Whilst blocks give you the ability to define localised input and output parameters, they do not affect the way that labels are scoped. Labels defined in blocks are not local to the scope of the block—this means it is possible to jump into and out of a block. Doing so will probably lead to disaster and abnormal termination of your application because the locals within a block have not been removed or have not been created.

Modules and Libraries

When applications grow large they are usually broken into multiple smaller, manageable pieces, commonly called **modules**. Each piece is assembled separately and then the pieces are stitched together by the linker to produce the final application.

When you partition a application into separate modules you will need to indicate how a symbol defined in one module is referenced in other modules. This section will show you how to declare symbols **exported** or **imported** so they can be used in more than one module.

Assembler features

The SmartDeck tools were designed to be flexible and let you to easily write space- efficient programs. To that end, the assembler and linker combination provides a number of features which are not found in many compilation systems.

Optimum-sized branches

The linker automatically resizes branches to labels where the label is too far away to be reached by a branch instruction. This is completely transparent to you as a programmer—when you use branch instructions, your linked program will always use the smallest possible branch instruction. This capability is deferred to the linker so that branches across compilation units are still optimised.

Fragments: killing dead code and data

The most important features of the linker are its ability to leave all unreferenced code and data out of the final application and to optimise the application as a whole, rather than on a per-function basis. The linker automatically discards all code and data fragments in a program that are not reachable from the start symbol.

Exporting symbols

Only symbols exported from a module can be used by other modules. You can export symbols using the `.EXPORT` directive. This directive does nothing more than make the symbol visible to other modules: it does not reserve storage for it nor define its type.

Syntax

`.EXPORT` **symbol** [, **symbol**]...

Not all symbols can be exported. Variables, labels, function blocks, and numeric constants defined using `EQU` can be exported, but macro names and local stack-based variables cannot.

The SmartDeck assembler publishes the symbol in the object file so that other modules can access it. If you don't export a symbol you can only use it in the source file it's declared in.

Using "::" is shorthand for defining and exporting labels

As a convenience, a label can be defined and exported at the same time using double- colon notation.

Example

```
data_ptr::
```

This declares the label `data_ptr` and exports it. This is equivalent to:

```
.EXPORT data_ptrdata_ptr:
```

Importing symbols

When you need to use symbols defined in other modules you must import them first. You import symbols using the `.IMPORT` directive.

Syntax

`.IMPORT` **symbol** [`::` **type**] [`,` **symbol** [`::` **type**]]...

When importing a symbol you can also define its type. This type information is used by the assembler whenever you reference the imported symbol and acts just like a symbol declared locally within the module. If you don't define a type for the imported variable, no type information is available to the assembler. If you subsequently use such a variable where type information is required, the assembler will report an error.

Example

```
.IMPORT CLA::BYTE, La::WORD.IMPORT APDUData::BYTE[256].IMPORT _myVar
```

The above imports `CLA` as a byte, `La` as a 16-bit word, `APDUData` as an array of 256 bytes, and `_myVar` without type information.

Other terminology for import and export

The terms import and export are not the only ones used to define symbol linkage between modules. If you've programmed in other assembly languages or used other assemblers before you may be more familiar with the terminology **public symbols** and **external symbols**. You can use the following synonyms for `.IMPORT` and `.EXPORT`:

Directive	Synonyms
`.IMPORT`	.EXTERN, XREF
`.EXPORT`	.PUBLIC, XDEF

Table 36 Import/export directive synonyms

Using libraries

When you know that you will need routines is a specific library you can use the `INCLUDELIB` directive to include them. Using `INCLUDELIB` means that you don't need to specify the library name on the compiler driver command line when you link your program.

Syntax

`INCLUDELIB "`**libname**`"`

libname is the name of the library you'd like the linker to include at link time. The above syntax is equivalent to the linker switch "`-l`**libname**."

Example

```
INCLUDELIB "c"
```

The above will cause the linker to search for and link the library `libc.hza`.

The SmartDeck compilers use this ability to transparently ask the linker to include the necessary run-time support package for the given language. The following libraries are automatically included when using object files produced by one of the SmartDeck compilers.

Library	Language
`libc.hza`	C
`libb.hza`	BASIC
`libf.hza`	Forth

Table 37 Language libraries linked automatically

Macros, conditions and Loops

Conditional assembly allows you to control which code gets assembled as part of your application, allowing you to produce variants. Macros and loops automate repetetive tasks, such as constructing tables or duplicating code.

Conditional assembly

The structure of conditional assembly is much like that used by high-level language conditional constructs and the C pre-processor. The syntax is:

Syntax

```
IF expression
        true-conditional-body
ENDIF
```

or

```
IF expression
        true-conditional-body
ELSE
        false-conditional-body
ENDIF
```

The controlling expression must be an absolute assembly-time constant. When the expression is non-zero the true conditional arm is assembled; when the expression is zero the false conditional body, if any, is assembled.

Nested conditionals

You'll find that using the `IF` and `ENDIF` directives on their own sometimes produces code which is difficult to follow.

Example

Consider the following which has nested IF directives:

```
IF type == 1
        CALL type1
ELSE
        IF type == 2
                CALL type2
        ELSE
                CALL type3
        ENDIF
ENDIF
```

The nested conditional can be replaced using the `ELIF` directive which acts like `ELSE IF`:

```
IF type == 1
        CALL type1
ELIF type == 2
        CALL type2
ELSE
        CALL type3
```



```
ENDIF
```

The full formal syntax for using `IF`, `ELIF`, and `ENDIF` is:

Syntax

```
IF expression
   statements
{ ELIF
   statements }
[ ELSE statements ]
ENDIF
```

Typical uses for conditional assembly

There are a few typical ways in which you can use the conditional assembly feature. Mostly these involve turning on or off a piece of code according to some criterion, for example to assemble it with debugging enabled, or to assemble it for a specific variant of operating system.

Omitting debugging code

When you write an application you'll invariable debug it by either using a debugger or by inserting special debugging code. When the final application is deployed, you'll want the application to be delivered with the debugging code removed—it takes up valuable resources and won't be necessary in the final application. However, it's always nice to keep the debugging code around just in case you should need to test the application again.

Example

Usual practice is to use a symbol, `_DEBUG`, as a flag to either include or exclude debugging code. Now you can use `IF` with the `DEFINED` operator to conditionally assemble some parts of your application depending upon whether the `_DEBUG` symbol is defined or not. You can define `_DEBUG` on the compiler driver or assembler command line.

```
IF DEFINED _DEBUG
   CALL DumpAppStateENDIF
```

You can control whether the call to `DumpAppState` is made by defining the symbol `_DEBUG`. You can do this by defining the symbol in the assembler source file, or more flexible would be to define the symbol when assembling the file by using the compiler driver's `-D` option.

```
hcl -c -D_DEBUG app.asm
```

To assemble the application for production you would leave the `_DEBUG` symbol undefined and assemble using:

```
hcl -c app.asm
```

Targeting specific environments

Another common use of conditional assembly is to include or exclude code according to the intended target operating system. For example, an application may need to be deployed on both MULTOS 3 and MULTOS 4 cards, but should take advantage of some extended features in MULTOS 4.

You can do this by defining a symbol and setting its value on the command line. Then, inside the application you can examine the value of this symbol to conditionally assemble a piece of code.

Example

A feature called ***transaction protection*** only exists in MULTOS version 4 and later; earlier versions of MULTOS do not support it. An application may need to be targeted to versions of MULTOS both with and without transaction protection, for example during a transition from one MULTOS card to another.
We can use the conditional assembly feature to target both MULTOS versions and take advantage of the transaction protection feature in version 4. The following code uses the value of `__MULTOS_VERSION` to conditionally assemble the calls to `TransProtOn` and `TransProtOff` which turn

transaction protection on and off for MULTOS 4 and later:

```
IF __MULTOS_VERSION >= 4
   CALL TransactionProtectionOn
ENDIF
INCN TransNum
IF __MULTOS_VERSION >= 4
   CALL TransactionProtectionOff

ENDIF
```

You set the specific version of MULTOS to target by defining the `__MULTOS_VERSION` symbol on the command line. To assemble the code for MULTOS version 3 and therefore exclude transaction protection you would use:

```
hcl -c -D__MULTOS_VERSION=3 trans.asm
```

And to assemble for MULTOS version 4 and include transaction protection you would use:

```
hcl -c -D__MULTOS_VERSION=4 trans.asm
```

Macros

The structure of a macro definition consists of a name, some optional arguments, the body of the macro and a termination keyword. The syntax you use to define a macro is:

Syntax

name `MACRO` ***arg***$_1$, ***arg***$_2$, ***...*** , ***arg***$_n$
macro-body
`ENDMACRO`

The name of the macro has the same requirements as a label name (in particular it must start in column one). The arguments are a comma-separated list of identifiers. The body of the macro can have arbitrary assembly language text including other macro definitions and invocations, conditional and file inclusion directives.

A macro is instantiated by using its name together with optional actual argument values. A macro instantiation has to occur on its own line—it cannot be used within an expression or as an argument to an assembly code mnemonic or directive. The syntax you use invoke a macro is

Syntax

name actual$_1$, ***actual***$_2$, ***...*** , ***actual***$_n$ / / ***comment***

When a macro is instantiated the macro body is inserted into the assembly text with the actual values replacing the arguments that were in the body of the macro definition.

Example

```
add    MACRO  N,a,b,c
       load   a, N
       load   b, N
       addn ,N
       pop N
       store c,N

ENDMACRO


longlong MACRO name, storage
name storage BYTE[8
ENDMACRO
 longlong A, STATIC
 longlong B, STATIC
 longlong C, STATIC
.......
```



```
add 8, A, B, C // C = A + B
```

This example shows two macro definitions: `add` which defines a code sequence that will add two variables and store the result in a third variable; and `longlong` defines an 8-byte variable.

Labels in macros

When labels are used in macros they must be unique for each instantiation to avoid duplicate label definition errors. The assembler provides a label generation mechanism for situations where the label name isn't significant and a mechanism for constructing specific label names.

Label generation

If a macro definition contains a jump to other instructions in the macro definition it is likely that the actual name of the label isn't important. To facilitate this a label of the form `name?` can be used

Example

```
incifeq MACRO a, N
  pushw N
  load a
  cmpn ,2
  bne NE?
  incn a
NE?

ENDMACRO
```

This example defines a macro `incifeq` that will increment a variable only if the variable has a defined value. For example

```
var1 STATIC WORD = 10
incifeq var1, 10
incifeq var1, 10
incifeq var1, 10
```

will increment `var1` to be 11. If the label in the macro didn't use a '`?`' then the assembler would have produced an error that the label was defined multiple times.

Label construction

There are situations when a macro invocation should result in the definition of a label. In the simplest case the label can be passed as an argument to the macro, however there are cases when the label name should be constructed from other tokens. The macro definition facility provides two constructs to enable this:

- tokens can be concatenated by putting ## between them;
- the value of a constant label can be used by prefixing the label with $$.

Example

```
MELBINARYOP macro opcode
  _mel##opcode##$$MELNUM BLOCK
  op1? IN WORD
  op2? IN WORD
ccr OUT WORD
  loadi op1?, $$MELNUM
  loadi op2?, $$MELNUM
  opcode ,$$MELNUM
  pop $$MELNUM
  storei op1?,$$MELNUM  prim 0x5  store ccr,1 RETURN
ENDMACRO
```

This macro will define a function whose name is based on a macro argument and the value of the label `MELNUM`. For example the macro invocations

```
MELNUM SET 3
MELBINARYOP addn
MELNUM SET MELNUM+1
MELBINARYOP addn
```

will create the functions `_meladdn3` and `_meladdn4`

Loops

If multiple definitions are required a loop structure can be used. This can be achieved either by a recursive macro definitions or by the use of the `LOOP` directive.

Example

```
MKMELARITH macro N
IF N  MKMELARITH (N-1)
  MELNUM SET MELNUM+1
  MELBINARYOP addn
ENDIF
ENDMACRO
MKMELARITH 10
```

Will produce functions `_meladdn10 ... _meladdn1`. If the loop counter is a large number then a recursive macro may consume considerable machine resources. To avoid this we recommend using the `LOOP` directive, which is an iterative rather than recursive solution.

Syntax

```
LOOP (expression)
       loop-body
ENDLOOP
```

The loop control expression must be a compile test constant. The loop body can contain any assembly text (including further loop constructs) except macro definitions (since it would result in multiple definitions of the same macro)

Example

```
x set 3
LOOP (x)
  pushb (x*y)
  x set x-1
ENDLOOP
```

This will assemble the instructions `pushb 3`, `pushb 2` and `pushb 1`. Note that the label naming capabilities (`? $$ ##`) are not available within the body of a loop. If the loop body is to declare labels then a recursive macro definition should be used or a combination of using macro invocation to define the labels and the loops to define the text of the label.

Mnemonic Reference

This section describes the MEL assembler mnemonics used by the SmartDeck tools. It is particularly useful if you need to write small MEL assembler routines to augment your application.

This document does not describe the AAM architecture in detail; refer to the ***MULTOS Application Programmers Reference Manual*** for further details.

Instructions

This section contains a summary of the syntax of MEL instructions.

ANDB

`ANDB` ***addr, byte***

`ADDB` **,** ***byte***

Perform a bitwise and of ***byte*** into the byte at address ***addr***. If ***addr*** is omitted, the source and destination is the byte on the top of the stack. This operation is a convenience and expands to the bit manipulate byte primitive when the operand is on the top of the stack and to a sequence of instructions when the operand is in memory.

ADDB

`ADDB` ***addr, byte***

`ADDB` **,** ***byte***

Add the 8-bit literal ***byte*** to the byte at address ***addr***.

ADDN

`ADDN` ***addr, len***

`ADDN` **,** ***len***

Add the top of stack item to the block at address ***addr***.

ANDW

`ANDW` ***addr, byte***

`ADDW` **,** ***byte***

Perform a bitwise and of ***word*** into the word at address ***addr***. If ***addr*** is omitted, the source and destination is the word on the top of the stack. This operation is a convenience and expands to the bit manipulate word primitive when the operand is on the top of the stack and to a sequence of instructions when the operand is in memory.

ADDW

`ADDW` ***addr, word***

`ADDW` **,** ***word***

Add the 16-bit literal ***word*** to the word at address ***addr***.

ANDN

`ANDN` ***addr, len***

`ANDN` **,** ***len***

Perform a bit-wise and of the data at tagged address ***addr*** and the top of stack item `{DT[-`***len***`],` ***len***`}` and place the result at ***addr***.

BRANCH

B***cond label***

BRA *label*

Transfer control to ***label*** if the condition codes meet the condition ***cond***. The condition ***cond*** is one of `EQ`, `LT`, `LE`, `GT`, `GE`, `NE`, or `A`.

CALL

`C`***cond label***

`CALL` ***label***

The first form calls the subroutine ***label*** if the condition codes meet the condition ***cond***. The condition ***cond*** is one of `EQ`, `LT`, `LE`, `GT`, `GE`, `NE`, or `A`. If no condition is specified, `A` is assumed.

CALL

The second form is an indirect call and expects a code address on the top of stack and calls that address.

CLEARN

`CLEARN` ***addr, len*** `CLEARN ,` ***len***

Clear the block at ***addr*** of length ***len*** bytes.

CMPB

`CMPB` ***addr, byte*** `CMPB ,` ***byte***

Compare the 8-bit literal ***byte*** with the byte at address ***addr***.

CMPN

`CMPN` ***addr, len*** `CMPN ,` ***len***

Add the top of stack item `{DT[-`***len***`], `***len***`}` to the block at address ***addr***.

CMPW

`CMPW` ***addr, word*** `CMPW ,` ***word***

Compare the 16-bit literal ***word*** to the word at address ***addr***.

DECN

`DECN` ***addr, len*** `DECN ,` ***len***

Decrement by one the block at ***addr*** of length ***len*** bytes.

Example

```
DECN , 2 ; decrement top of stack by 2 DECN SB[0], 1 ; decrement the byte at
SB[0] by 1 DECN x ; decrement x. Size is derived from x's type
```

EXIT

`EXIT`

Return control to MULTOS.

EXITLA

`EXITLA` ***la1, la2***

`EXITLA` ***la***

Set La to ***la1***:***la2*** or ***la*** as appropriate and return control to MULTOS.

Example

```
EXITLA $00, $12
EXITLA $12
```


Both the above instructions assemble to identical code. The second is the preferred form.

EXITSW

```
EXITSW sw1, sw2
EXITSW sw12
```

Set status word to ***sw1***, ***sw2*** or ***sw12*** as appropriate and return control to MULTOS.

Example

```
EXITSW $6e, $00
EXITSW $6e00
```

Both the above instructions assemble to identical code. The second is the preferred form.

EXITSWLA

```
EXITSWLA sw1, sw2, la
EXITSWLA sw12, la
```

Set status word to ***sw1***, ***sw2*** or ***sw12*** as appropriate, set La to ***la***, and return control to MULTOS.

Example

```
EXITSWLA $6e, $00, 0
EXITSWLA $6e00, 0
```

Both the above instructions assemble to identical code. The second is the preferred form.

EQVB

```
EQVB , byte
```

Perform a bitwise equivalence of ***byte*** and the byte on the top of the stack. This operation is a convenience and expands to the bit manipulate byte primitive.

EQVW

```
EQVW , word
```

Perform a bitwise equivalence of ***word*** and the word on the top of the stack. This operation is a convenience and expands to the bit manipulate word primitive.

INCN

```
INCN addr, len
INCN , len
```

Increment by one the block at ***addr*** of length ***len*** bytes.

INDEX

```
INDEX addr, size
```

Push the segmented address formed by multiplying the byte `{DT[-1], 1]` by ***size*** and adding it to ***addr***. The multiplication is an unsigned 8-bit by 8-bit multiplication.

JUMP

```
Jcond label
JMP label
JMP ,
```

The first form transfers control to ***label*** if the condition codes meet thecondition ***cond***. The condition ***cond*** is one of `EQ`, `LT`, `LE`, `GT`, `GE`, `NE`, or `A`.The second form is an indirect jump and expects a code address on the top of stack and transfers control to that address.

LOAD

```
LOAD addr, len
LOAD , len
```

Push onto the stack the block at ***addr*** of length ***len*** bytes.

LOADA

```
LOADA addr
```

Push onto the stack the segmented address corresponding to the tagged address ***addr***.

LOADI

```
LOADI addr, len
LOADI , len
```

Load indirect onto the stack. The word {***addr***, 2}contains the segment address of the block to push and ***len*** is the block length.

NOTN

```
NOTN addr, len
NOTN , len
```

Complement the block at ***addr*** of length ***len*** bytes.

ORB

```
ORB addr, byte
ORB , byte
```

Perform a bitwise or of ***byte*** into the byte at address ***addr***. If ***addr*** is omitted, the source and destination is the byte on the top of the stack. This operation is a convenience and expands to the bit manipulate byte primitive when the operand is on the top of the stack and to a sequence of instructions when the operand is in memory.

ORN

```
ORN addr, len
ORN , len
```

Bit-wise inclusive-or the top of stack item with the block at address ***addr*** and place the result at ***addr***.

ORW

```
ORW addr, byte
ORW , byte
```

Perform a bitwise or of ***word*** into the word at address ***addr***. If ***addr*** is omitted, the source and destination is the word on the top of the stack.This operation is a convenience and expands to the bit manipulate word primitive when the operand is on the top of the stack and to a sequence of instructions when the operand is in memory.

PRIM

```
PRIM prim
PRIM prim, b1
PRIM prim, b1, b2
PRIM prim, b1, b2, b3
```

Call the primitive ***prim*** in set 0, 1, 2, or 3 with the given arguments. Refer to the ***MULTOS Application Programmers Reference Manual*** for further information.

POPB


```
POPB
```

Pop one byte from the stack and discard it.

POPW

```
POPW
```

Pop one word from the stack and discard it.

POPN

```
POPN len
```

Pop ***len*** bytes from the stack and discard them.

PUSHB

```
PUSHB byte
```

Push the 8-bit literal ***byte*** onto the stack.

PUSHW

```
PUSHW word
```

Push the 16-bit literal ***word*** onto the stack.

PUSHZ

```
PUSHZ len
```

Push a block of ***len*** zeroes onto the stack.

RET

```
RET input-len, output-len
```

Return ***output-len*** bytes on the stack as the result of the subroutine call and deallocate ***input-len*** bytes from the stack on return.

SETB

```
SETB addr, byte
SETB , byte
```

Set the byte at address ***addr*** to the 8-bit literal ***byte***.

SETLA

```
SETLA la1, la2
SETLA la
```

Set La to ***la1***:***la2*** or ***la*** as appropriate.

Example

```
SETLA $00, $12
SETLA $12
```

Both the above instructions assemble to identical code, but the second is the preferred form.

SETSW

```
SETSW sw1, sw2
```



```
SETSW sw12
```

Set status word to ***sw1***, ***sw2*** or ***sw12*** as appropriate.

Example

```
SETSW $6e, $00
SETSW $6e00
```

Both the above instructions assemble to identical code. The first is allowed for MDS compatibility, but the second is the preferred form.

SETSWLA

```
SETSWLA sw1, sw2, la
SETSWLA sw12, la
```

Set status word to ***sw1***, ***sw2*** or ***sw12*** as appropriate, set La to ***la***.

Example

```
SETSWLA $6e, $00, 0
SETSWLA $6e00, 0
```

Both the above instructions assemble to identical code, but the second is the preferred form.

SETW

```
SETW addr, word
SETW , word
```

Set the word at address ***addr*** to the 16-bit literal ***word***.

STORE

```
STORE addr, len
STORE , len
```

Store the top of stack to the block at ***addr*** of length ***len*** bytes and pop ***len*** bytes from the stack.

STOREI

```
STOREI addr, len
STOREI , len
```

Store indirect from the stack. The word at `{`***addr***`, 2}` contains the segment address of the block to store the top of stack in and ***len*** is the block length. Once stored, ***len*** bytes are popped from the stack.

SUBB

```
SUBB addr, byte
SUBB , byte
```

Subtract the 8-bit literal ***byte*** from the byte at address ***addr***

SUBN

```
SUBNN addr, len
SUBN , len
```

Subtract the top of stack item from the block at address ***addr***.

SUBW

```
SUBW addr, word
```



```
SUBW , word
```

Subtract the 16-bit literal ***word*** from the word at address ***addr***.

SYSTEM

```
SYSTEM op
SYSTEM op,w1
SYSTEM op,w1, w3
```

Perform the given system action. Please refer to the ***MULTOS Application Programmers Reference Manual*** **for further information.**

TESTN

```
CLEARN addr, len
CLEARN , len
```

Test the block at ***addr*** of length ***len*** bytes against zero.

XORB

```
XORB addr, byte
XORB , byte
```

Perform a bitwise exclusiv -or of ***byte*** into the byte at address ***addr***. If ***addr*** is omitted, the source and destination is the byte on the top of the stack.This operation is a convenience and expands to the bit manipulate byte primitive when the operand is on the top of the stack and to a sequence of instructions when the operand is in memory.

XORN

```
XORN addr, len
XORN , len
```

Exclusive or the top of stack item into the block at address ***addr***.

XORW

```
XORW addr, byte
XORW , byte
```

Perform a bitwise exclusive or of ***word*** into the word at address ***addr***. If ***addr*** is omitted, the source and destination is the word on the top of the stack. This operation is a convenience and expands to the bit manipulate word primitive when the operand is on the top of the stack and to a sequence of instructions when the operand is in memory.

C User Guide

SmartDeck C is a faithful implementation of the ANSI and ISO standards for the programming language C. This manual describes the C language as implemented by the SmartDeck C compiler.

Because smart cards are small, memory-limited devices, we have to make a few concessions. For instance, the ANSI C input/output library is much too large for a smart card environment, so this implementation of C simple does not provide it.

Preprocessor

The SmartDeck C preprocessor provides a number of useful facilities which extend the underlying compiler. For instance, the preprocessor is responsible for finding header files in `#include` directives and for expanding the macros set using `#define`.

In many implementations the C preprocessor is a separate program from the C compiler. However, in SmartDeck C the preprocessor is integrated into the C compiler so that compilations are faster.

Data types

This section defines the data type format used by the SmartDeck C compiler.

Byte order

All data items are held in the native big-endian byte order of the MULTOS AAM.

Plain characters

The plain character type is unsigned by default.

Floating-point types

The floating-point types `float` and `double` are not supported.

Type sizes

Data type	Size in bytes	Alignment in bytes
`char`	1	1
`unsigned char`	1	1
`int`	2	1
`unsigned int`	2	1
`short`	2	1
`unsigned short`	2	1
`long`	4	1
`unsigned long`	4	1
`long long`	**8**	**1**
`unsigned long long`	**8**	**1**
`float`	***not supported***	***not supported***
`double`	***not supported***	***not supported***
`type *` `(pointer)`	2	1
`enum (enumeration)`	2	1

Table 38 Data type sizes

SmartDeck C and the MULTOS environment

The SmartDeck C compiler integrates well with MULTOS, but there are a few things that you should know when using the C compiler on MULTOS.

Differences between MULTOS implementations

Some features of MULTOS are designated ***optional*** which means that a MULTOS implementation may or may not support a particular feature. Testing your code on the card you are deploying onto, therefore, is vital because the SmartDeck MULTOS simulator may support optional primitives that the card does not..

The features that a card supports are described in the ***MULTOS Implementation Report*** document which is available from the MAOSCO web site. You must register as an application developer to gain access to this document and other technical resources on the MULTOS web site.

Session data and stack sizes

The size of dynamic data varies between MULTOS implementations, so you should check that your application works on the card that you wish to deploy onto. To check the amount of session and dynamic data available on a card, consult the ***MULTOS Implementation Report*** which is available from the MULTOS website.

Optional support for MULTOS N and V flags

The N and V flags of the MULTOS condition code register are designated as optional flags. The SmartDeck compiler generates code that does not rely on the MULTOS card supporting signed arithmetic and as such does not use the N and V flags. All library routines are written so that they will execute correctly on cards which do not support signed arithmetic.

You can safely deploy your code on any card whether it supports signed arithmetic or not, fully confident that it will work correctly.

Optional MULTOS Primitives

Some MULTOS primitives are designated optional, such as certain crytographic primitives and transaction protection. It is impossible for the compiler to check that the primitives you use are available on the card you deploy onto. If you use a primitive that is not supported by the card, your application will abend.

Atomicity and data item protection

You cannot rely on the code generator of the compiler to write atomically to a variable.

Example

As an example, consider the following program fragment:

```
static long var;
void inc_var(void
{
  var += 2;

}
```

The compiler uses two instructions to increment the variable:

```
    INCN SB[var], 4
    INCN SB[var], 4
```

If you want to use atomic writes and data item protection you should always check the code generated by the compiler for your critical routines. Rather than using data item protection, you should consider using transaction protection which is easier to use than data item protection in high-level languages such as C.

User flags

The compiler and runtime system do not use the four flags in the condition code register set aside for the application to use. You may use these flags in assembler without having them changed by compiler code or the runtime syhstem.

Selecting where data are stored

The compiler currently supports four pragmas for data placement,

```
#pragma melstatic

#pragma melperso#pragma melpublic

#pragma melsession
```

These directives control where subsequent data ***definitions*** are generated. For instance, to place an integer `x` into the public section you would use:

```
#pragma melpublic

int x;
```

The pragma remains in effect until another section pragma is seen. By default all global data are placed into the MULTOS static section. It is not necessary to use these pragmas in header files when ***declaring*** external variables. For example the following is enough to make `my_session_data` available to other compilation units by using a header file:

```
#ifndef session_H
#define session_H
/* Session data *
extern int my_session_data;
#endif
```

One compilation unit must define `my_session_data`, and this is the place where the section pragma is used:

```
#include "session.h"
#pragma melsession
int my_session_data;
```

It's wise to switch back to the static segment afterwards so that further definitions don't go into the session data segment by accident. For example, the above is better written:

```
#include "session.h"
#pragma melsession

int my_static_data;
#pragma melstatic
```

#pragma melstatic

This pragma instructs the compiler to generate all further data into the "`.SB`" section, which is where MULTOS static data are held.

#pragma melperso

This pragma instructs the compiler to generate all following data into the beginning of the"`.SB`" section. Only one of these directives should be used in an application. It allows data that is to be personalised to be placed in a known location that will not change between versions of the application. The command **hls –perso <hzxfile>** outputs the names and locations of data items in this section.

#pragma melpublic

This pragma instructs the compiler to generate all further data definitions into the "`.PB`" section, which is

where MULTOS public data are held.

#pragma melsession

This pragma instructs the compiler to generate all further data definitions into the "`.DB`" section, which is where MULTOS session data are held.

Bytecode Substitution for Function Calls

The 'C' compiler and linker support a mechanism whereby a function can be implemented by substituting the CALL instruction for a set of MEL instructions defined statically in a BYTE array. This makes the repeated use of small functions much more efficient as no actual function call is made. It is similar to using a #define macro but is cleaner and makes full use of the Eclipse IDEs type checking functionality.

Function prototypes should be declared as normal. The compiler will manage the pushing of parameters and the popping and setting of return values. The bytecode for the function must be defined as in the following examples from C-API V2.

```
#define SET0PRIM 0x28
#define SET1PRIM 0x29
#define SET2PRIM 0x2A
#define PARAM_MARKER 0xFF

#define DO_AND 0x83
#define ZFLAG 0x01

#define ZFLAG_SET SET0PRIM,PRIM_LOAD_CCR,SET2PRIM,__PRIM_BIT_MANIPULATE_BYTE,DO_AND,ZFLAG

// Prototype
BOOL multosBCDtoBIN (BYTE *sourceAddr, BYTE *destAddr, BYTE sourceLen, BYTE destLen);
BOOL multosCardBlock (const BYTE MSB_StartAddress_MAC, const BYTE LSB_StartAddress_MAC);

#pragma melbytecode
BYTE _bytecode_multosBCDtoBIN[] = {SET1PRIM, __PRIM_CONVERT_BCD, 0x00, ZFLAG_SET };
BYTE _bytecode_multosCardBlock[] = { SET2PRIM, __PRIM_CARD_BLOCK, PARAM_MARKER, PARAM_MARKER,
ZFLAG_SET};
```

If the first instruction in the array is a *primitive* call and that primitive has fixed parameters, it is possible to set those values (still at compile time) but using single BYTE parameters that *look like* they are being passed via the stack. These are the parameters shown as **const** in the examples above. Their value gets put into the parameter placeholders marked by a 0xFF value (PARAM_MARKER in the example above).

Assembler inserts

You can place assembler code directly into the output of a function using the `__push`, `__pop`, and `__code` statements. SmartDeck software provides efficient access to MULTOS services through the use of inline assembly code. The statements `__push()`, `__code()` and `__pop()` enable C expressions to be used in MEL instructions and MEL instructions to be embedded within a C program. When using inline assembly code it is vitally important to keep the stack balanced: don't take too much data off the stack, and don't leave data on the stack after an assembler insert.

Pushing values onto the stack

The `__push` statement takes a C expression as an argument and pushes the result of that expression onto the MEL execution stack.

Syntax

`__push(`***expression***`, ***expression...***`);`

Example

```
int x = 5;__push(3, x);__push(&x);
```

This generates:

```
pushw 3load x, 2loada x
```

which loads the literal word 3, the integer value of `x`, and the address of `x` onto the stack.

Inserting instructions

You can place a specific MEL instruction into a program using the `__code` statement.

Syntax

`__code(`***instruction*** [`,` ***operand***$_1$ [`,` ***operand***$_2$[`,` ***operand***$_3$]]]`);`

Each instruction that is supported is defined in the file `<melasm.h>`. The following instructions can be used with `__code`:

- `LOAD, STORE, LOADI, STOREI, LOADA, INDEX`
- `SETB, CMPB, ADDB, SUBB`
- `SETW, CMPW, ADDW, SUBW`
- `CLEARN, TESTN, INCN, DECN, NOTN`
- `CMPN, ADDN, SUBN, ANDN, ORN, XORN`
- `SYSTEM, PRIM, POPB, POPW, PUSHB, PUSHW, PUSHZ`

The following instructions cannot be used with `__code` because they affect the code generation strategy of the compiler and the optimizer:

- `CALL, RET, BRANCH, JUMP`

You can include addresses and sizes in instructions.

Example

```
__code(ADDN, 4)
__code(ADDN, &x, 4);
```

This generates:

```
addn , 4
addn x, 4
```

Note that the address given must be a directly addressable location, that is the address of a local variable, a parameter, or a global.

Storing values to memory

The `__pop` statement must be given an expression that evaluates to an address and stores the number of bytes corresponding to the size of the expression.

Syntax

`__pop(`***expression***`, ***expression...***`);`

Example

```
struct
```



```
{
  int f1;
  int f2;
} x;

__pop(&x)
```

This generates

```
store x,4
```

which will store four bytes from the stack into the address denoted by `x`.

Assembler Interfacing

Using SmartDeck it's possible to write mixed-language programs as all SmartDeck compilers and assemblers share an identical object and debugging format. SmartDeck has been designed to make working with multiple languages as simple as possible— but as with any multi-language program, its going to take a little effort from you, the user, to make everything run smoothly.

The SmartDeck C compiler provides a substantial library in order to interface with MULTOS, but there may be times when you will need to interface C with assembler code—for instance, because you have legacy code already written in assembler, or you need to code in assembler because there are some things which can't be done using a high-level language. For this you will need to know the conventions which the C compiler uses to name variables, how parameters are passed on the stack, and how results are returned to name but a few.

In the following sections we'll examine how the SmartDeck C compiler can work with assembly code routines. You'll also find multi-language examples in the software distributed to you.

Naming conventions

SmartDeck C prefixes all C names with an underscore at the assembler level. Thus, variable names do not clash with type names or assembler mnemonics. The C compiler translates the declaration

```
int myVar = 1;
```

into the following code:

```
_myVar: DW 1
```

This convention is used for all globally scoped variables and functions. Locally scoped automatic variables and parameters are not given names in assembler, only addresses relative to the frame pointer.

Calling conventions

You should find the calling conventions for the SmartDeck compiler quite natural, especially given the architecture of the MULTOS virtual machine. In the following sections you'll see the rules that SmartDeck uses to pass parameters and to return results.

How parameters are passed

Parameters are passed using the standard convention of C compilers: reverse order, last parameter pushed first. As an example, consider the following function declaration:

```
void fn(int x, int y,int z);
```


When calling this function, the caller pushes the value for `z` first, `y` second, and `x` last, so calling

```
fn(1, 2, 3)
```

generates:

```
PUSHW 3
PUSHW 2
PUSHW 1
CALL _fn
```

Parameters are pushed onto the stack using a caller-widening scheme, that is the caller widens or narrows an actual parameter to the appropriate size for the formal parameter defined in the function prototype.

Example

```
void fn(int x);
long a;

void main(void)
{
  fn(a);
}
```

Here, the caller narrows the 32-bit value of `a` to the size expected by the caller, namely 16 bits:

```
LOAD _a, 4
STORE , 2
CALL _fn
```

Naturally, the caller must also extend any value which is too narrow:

```
void fn(int x);
unsigned char a;

 void main(void
 {
   fn(a);
 }
```

Here, the caller extends the 8-bit value of `a` to the size expected by the caller, namely 16 bits:

```
PUSHB 0
LOAD _a, 1
CALL _fn
```

Cleaning the stack on return

It is the called function's responsibility to clean the stack when it returns. For fully prototyped functions this is possible by encoding the parameter area's size in bytes in the `RET` instruction.

How values are returned

Values are returned to the caller on the stack according to the function's type declared in the prototype. For example, if a function is declared to return a `char`, one byte must be returned by the caller to the callee rather than promoting the character to an `int`.

Using assembler directives for interfacing

In the following sections we will put into practice the conventions outlined above by showing how to interface C with assembler using the appropriate directives. As SmartDeck provides the `IN`, `OUT`, and `STACK` directives, interfacing C to assembler is considerably simplified.

Translating a simple prototype

Translating the C function prototype which takes a single parameter to a set of assembler directives is very simple. Consider the following simple prototype:

```
int add1(int x);
```


We will write an implementation of `add1` in assembler which adds one to the parameter `x` and returns it as the function result. The SmartDeck C compiler represents an `int` in a 16-bit word, and we translate the above prototype to:

```
_add1 BLOCK
x IN WORD
result OUT WORD
```

Note that the function return value is named, in this case `result`. When you want to return a value from the function, simply assign it to `result` and return; the assembler takes care of generating the correct set of instructions to do this. So, the body of `add1` is coded:

```
LOAD x
ADDW , 1
STORE result
```

The assembler derives the correct instruction length information for each instruction from the declarations we made earlier. If we needed to be explicit we could have used

```
LOAD x, 2
ADDW , 1
STORE result, 2
```

To return to the caller, you use `RETURN` directive which emits the correct code to clean up the stack and return to the caller:

```
LOAD x
ADDW , 1
STORE result
RETURN
```

So, in full the function `add1` is:

```
_add1  BLOCK
x      IN     WORD
result OUT    WORD
       LOAD   x
       ADDW , 1
       STORE result
       RETURN
```

Translating a prototype with more than one parameter

When translating a prototype with more than one parameter you must remember that the SmartDeck compiler pushes parameters in ***reverse order***. As such, if you have a function which takes parameters `x` and then `y`, you must declare `y` first in the assembler block and then `x`.

Example

Consider translating

```
int max3(int x, int y, int z);
```

which is supposed to return the maximum value of its three parameters. You must construct the assembler declaration which correctly interfaces with the SmartDeck C code as follows:

```
_max3 BLOCK
z IN WORD
y IN WORD
x IN WORD
result OUT WORD
```


C Library Reference

This section describes the functions available from the original C library of SmartDeck. Wherever possible the C-API as defined in multos.h should be used in preference as it provides access to all the latest features and make use of the most efficient primitives. The original library is still included to support backwards compatibility with older applications.

The library provides:

- multi-precision unsigned arithmetic in <multosarith.h>
- access to the CCR register in <multosccr.h>
- cryptographic primitives in <multoscrypto.h>
- controlling communications with the IFD in <multoscomms.h>
- other MAOS services in <multosmisc.h>
- a simulator-only printf capability in <stdio.h>
- non-local jump support in <setjmp.h>
- memory functions in <string.h>
- heap management functions in <stdlib.h> and <heap.h>
- DES and RSA encryption in <DES.h> and <RSA.h>
- standard C definitions in <stddef.h> and <limits.h>

Using the C library

Conventions

The C library contains a number of macros, functions, and variables. In general, we use the following conventions:

- Identifiers which are completely upper-case or have initial characters which are upper-case are macros.
- Identifiers which are in lower-case are true functions. However, there are a number of important exceptions.

Well-known data items such as the INS and CLA bytes of an APDU keep their ISO/EMV names and are variables whose names are all capitals (in this case `INS` and `CLA`).

Library implementation

Many of the routines in the library are implemented as macros that generate inline code. We have taken this approach because supporting many MULTOS primitives is impossible using a general-purpose assembler subroutine where the size of some data object is known only at run time—for MULTOS routines, the size of the object must be known at compile time and encoded directly into the instruction. See the header files for a description of each function / macro.

Cryptographic functions

Note: Please use the C-API as defined by multos.h for new applications.

The header file `<multoscrypto.h>` contains interfaces to the MULTOS cryptography functions. These include DES encipherment and decipherment, signature functions, modular multiplication for RSA, SHA-1, and asymmetric hash algorithms.

APDU control

Note: Please use the C-API as defined by multos.h for new applications.

The header file `<multoscomms.h>` contains variables and macros which allow you to retrieve C-APDU parameters and data and set R-APDU parameters and data.

Arithmetic functions

Note: Please use the C-API as defined by multos.h for new applications.

The header file `<multosarith.h>` contains macros which provide multi-precision arithmetic from C. The first parameter to all macros is the size of the block (N) which must be a compile- time constant.

Binary operations

For binary operations two forms of macro are supported:

- one which will update a variable as a result of the operation, and
- one which will store the result into another variable.

Examples

```
unsigned char x[1] = { 3 };
unsigned char y[1] = { 4 };
ASSIGN_ADDN(1, x, y);
```

This will add `y` to `x` which results in `x[0]` being set to 7.

```
unsigned char x[1] = { 3 };
unsigned char y[1] = { 4 };
unsigned char z[1];
ADDN(1, z, x, y);
```

This will add `x` to `y` and put the result in `z`.

Unary operations

For unary operations the variable argument will be updated.

Example

```
CLEARN(1, z)
```

will set `z` to zero. The block parameters to the macros will always be cast to be an N-byte array of unsigned characters.

Manipulating the CCR

Note: Please use the C-API as defined by multos.h for new applications.

The header file `<multosccr.h>` contains macros LoadCCR and StoreCCR to get and set the MULTOS condition code register (CCR).

MULTOS operating system functions

Note: Please use the C-API as defined by multos.h for new applications.

The header file `<multomisc.h>` contains macros that provide access to the MULTOS operating system functions (MEL primitives) from C.

DES library functions

Note: Please use the C-API as defined by multos.h for new applications.

Various flavours of DES enciphering are supported in the library <des.h>. These

include both ECB and CBC encryption modes, single key DES and 2 or 3 key triple DES. The triple DES encryption uses encrypt-decrypt-encrypt mode. The triple key CBC encryption uses Outer-CBC mode.

These functions operate on arbitrary sized messages and have variants that support no padding or user defined padding schemes to be specified. User defined padding is accomplished by supplying a function pointer that is called by the enciphering function. A pad function takes the number of padding bytes required and a pointer to the buffer and should pad that buffer appropriately.

```
typedef void (*padfntype)(unsigned numPadBytes,unsigned char *padBuffer);
```

The converse operation is to determine the number of padding bytes when the message is deciphered. An unpad function is passed a pointer to the last 8 byte block of the message and is expected to return the number of padding bytes in the block.

```
typedef unsigned(*unpadfntype)(unsigned char *padBuffer);
```

On decipher functions the padding bytes are not copied.

RSA library functions

Note: Please use the C-API as defined by multos.h for new applications.

The RSA library <RSA.h> defines structures for RSA public and private keys and supports functions that encipher and decipher using these keys. Before including the rsa header file the macros `MODLEN` and `EXPLEN` must be defined. For example

```
#define MODLEN 128
#define EXPLEN 2
#include <rsa.h>
```

will set the modulus length to be 128 bytes and the exponent length to be 2 bytes. The library requires that the RSA private key is in CRT form as defined by the following structure.

```
typedef struct
{
    unsigned modlenInBytes;
    unsigned char dp[MODLEN/2];
    unsigned char dq[MODLEN/2];
    unsigned char p[MODLEN/2];
    unsigned char q[MODLEN/2];unsigned char u[MODLEN/2];
} RSAPrivateCRTKey;
```

The RSA public key is defined with the following structure

```
typedef struct
{
    unsigned modlenInBytes;
    unsigned explenInBytes;
    unsigned char m[MODLEN];
    unsigned char e[EXPLEN];
} RSAPublicKey;
```

These structures can be used to define constant keys for example

```
RSAPublicKey publicKey =
{
    128, // modlenInBytes
    2, // explenInBytes
    { // m[128]
          0xda, 0x99, 0x7d, 0x0a, 0x44, 0xd5, 0x56, ....
          ...., 0x75, 0x95, 0x92, 0x6a, 0x6a, 0x37, 0x1d
    },
    { // e[2]
```

```
            0xc3, 0x53
        }
    };
```

Note that the sizes of the modulus and exponent may be restricted on the MULTOS implementation you are using.

Standard C library functions

A limited number of standard C library functions are supported in the library of SmartDeck C. Consult a C programming manual for details on how to use these functions that are detailed below.

<ctype.h>

This file contains implementation defined constants.

<heap.h>

This is not a standard C library; it is required to set up the heap used to implement the functions in `<stdlib.h>`

Prototype

```
void heap_init(size_t size, void *ptr);
unsigned heap_used(void);
```

Description

The `heap_init` function takes a block of memory of `size` bytes pointed to by `ptr`. This block of memory is used to implement the `<stdlib.h>` functions. The `heap_used` function returns the number of bytes of memory that have been used.

<limits.h>

This file contains functions for character testing.

<setjmp.h>

The standard functions `setjmp` and `longjmp` are supported in this library.

<stdio.h>

The standard function `printf` is supported by this library when running on the SmartDeck simulator. Use #ifdef statements to exclude from card builds.

<stdlib.h>

The standard heap functions `calloc`, `malloc`, `realloc` and `free` are supported by this library together with the math functions `abs`, `labs`, `div` and `ldiv`.

<string.h>

The standard string and memory functions are supported in this library. The functions `strtok` and `strerror` are not supported.

C Compiler Diagnostics

Pre-processor warning messages

These warning messages come from the pre-processing pass of the compiler. Although the compiler and pre-processor are integrated into the same executable, it is worth distinguishing the pre-processor warning messages from those generated by the compiler proper.

`bad digit 'digit' in number`
When evaluating a pre-processor expression the pre-processor encountered a malformed octal, decimal, or hexadecimal number.

`bad token 'token' produced by ##`
A bad pre-processing token has been produced when using the token pasting operator `##`. This error is extremely unlikely to occur in your code.

`character constant taken as not signed`
Characters with ASCII codes greater than 127 are treated as unsigned numbers by the pre-processor.

`end of file inside comment`
The pre-processor came to the end of file whilst processing a comment. This is usually an error: comments cannot extend across source files.

`multi-byte character constant undefined`
Multi-byte character constants are not supported by the pre-processor when evaluating expressions.

`no newline at end of file`
The last character in the file is not a new line. Although this isn't an error, it may help portability of your code if you include a new line at the end of your file.

`syntax error in #if/#endif`
There's a general problem in the way you've used the `#if` or `#endif` control

`unknown pre-processor control 'control'`
The pre-prpocessor control `#`***control*** isn't a valid ANSI pre-processor control. Usually this is caused by a spelling error.

`undefined escape '\char' in character constant` When evaluating a pre-processor expression the pre-processor encountered an escape sequence `\`***char*** which isn't defined by the ANSI standard.

`wide character constant undefined`
Wide character constants are not supported by the pre-processor when evaluating expressions.

Pre-processor error messages

These error messages come from the pre-processing pass of the compiler. Athough the compiler and pre-processor are integrated into the same executable, it is worth distinguishing the pre-processor error messages from those generated by the compiler proper.

`# is not followed by a macro parameter`

The `#` concatenation operator must be followed by a macro parameter name.

`## occurs at border of replacement`

The `##` operator cannot be placed at the end of a line.

`#defined token is not a name`

The token defined immediately after `#define` is not a valid pre processor identifier.

`#defined token '`*`token`*`' can't be redefined`

You cannot redefine a number of standard tokens such as `__LINE__` and `__STDC__`. The token you're trying to redefine is one of these.

`bad ?: in #if/#elif`

There is an error parsing the ternary `?:` operator in an expression. This is usually caused by mismatched parentheses or forgetting one of the `?` or `:` separators.

`bad operator '`*`operator`*`' in #if/#elif`

The operator operator is not allowed in pre-processor expressions.

`bad syntax for 'defined'` The `defined` standard pre-processor function does not conform to the syntax `defined(`***`name`***`)`.

`can't find include file '`*`file`*`'`

The include file file can't be found in any of the directories specified in compilation.

`disagreement in number of macro arguments to '`*`name`*`'`

The macro ***name*** has been invoked with either too few or too many actual arguments according to its formal argument list.

`duplicate macro argument '`*`name`*`'`

The macro argument ***name*** has been given twice in the argument list of a `#define` pre-processor control.

`end of file in macro argument list`

The pre-processor encountered the end of file whilst processing the argument list of a macro.

`illegal operator * or & in #if/#elif.`

The pointer dereference operator `*` and the address-of operator `&` cannot be used in pre-processor expressions.

`insufficient memory`

The pre-processor has run out of memory. This is a very unlikely error message, but if it does occur you should split up the file you are trying to compile into several smaller files.

`macro redefinition of '`*`name`*`'`

The macro ***name*** has been defined twice with two different definitions. This usually occurs when two header files are included into a C source file and macros in the header files clash.

`pre-processor internal error: `*`cause`*

The pre-processor has found an internal inconsistency in its data structures. It would help us if you could submit a bug report and supporting files which demonstrate the error.

`stringified macro argument is too long`

The stringified macro argument is longer than 512 characters. This error is unlikely to occur in user code and it isn't practical to show an example of this failure here.

syntax error in #ifdef/#ifndef

The pre-processor found an error when processing the `#ifdef` or `#ifndef` controls. This is usually caused by extra tokens on the pre-processor control line.

syntax error in #include

The pre-processor found an error when processing the file to include in an `#include` directive. The usual cause of this is that the file name isn't enclosed in angle brackets or quotation marks, or that the trailing quotation mark is missing.

syntax error in macro parameters

The syntax of the comma-separated list of macro parameters in a `#define` pre- processor control is not correct. This can occur for a number of reasons, but most common is incorrect puctuation.

undefined expression value

The pre-processor encountered an error when evaluating an expression which caused the expression to be undefined. This is caused by dividing by zero using the division or modulus operators.

unterminated string or character constant

A string is not terminated at the end of a line.

Compiler warning messages

'*function*' is a non-ANSI definition

You have declared the `main` entry point using an old-style function definition. `main` should be an ANSI-prototyped function. The compiler only reports this warning when extra-picky ANSI warnings are enabled.

Old-style function definitions, although valid, should not be used because they are a common source of errors and lead to code which is less efficient than a prototyped function. A function which takes no parameters should be declared with the parameter list `(void)`.

'*type*' is a non-ANSI type

The `unsigned long long` type is not supported by ANSI C. The SmartDeck C compiler supports this type as you would expect.

'*type*' used as an lvalue

You used an object of type ***type*** as an lvalue. Assigning through an uncast, dereferenced `void*` pointer is an error and will result in this error.

empty declaration

A declaration does not declare any variables (or types in the case of `typedef`).

The input file contains no declarations and no functions. A C file which contains only comments will raise this warning.

local '*type name*' is not referenced

The local declared variable ***name*** is not referenced in the function.

Compiler error messages

'*number*' is an illegal array size

Array sizes must be strictly positive — the value ***number*** is either zero or negative.

'*number*' is an illegal bit-field size

The size of a bit field be within the range 0 to `8*sizeof(int)`. In many cases this means that `long` data items cannot be used in bit field specifications.

'*type*' is an illegal bit-field type
The type of a bit field must be either an unsigned integer or a signed integer, and ***type*** is neither of these. Note that enumeration types cannot be used in bit fields.

'*type*' is an illegal field type
You cannot declare function types in structures or unions, only pointers to functions.

'(' expected
An opening parenthsis was expected after the built-in function `__typechk`.

addressable object required
An addressable object is required when applying the address-of operator '`&`'. In particular, you can't take the address of a simple constant.

assignment to const identifier '*name*'
You cannot assign to `const`-qualified identifiers.

assignment to const location
You cannot assign through a pointer to a `const`-qualified object nor can you assign to members of `const`-qualified structures or unions.

bad hexadecimal escape sequence '\x*char*'
The character ***char*** which is part of a hexadecimal escape sequence isn't a valid hexadecimal digit.

'break' not inside loop or switch statement
You have placed a `break` statement in the main body of a function without an enclosing `for`, `do-while`, `while-do`, or `switch` statement. This usually happens when you edit the code and remove a trailing close brace by mistake.

cannot initialize undefined '*type*'
You cannot initialize an undefined structure or union type. Undefined structure or union types are usually used to construct recursive data structures or to hide implementation details. They are introduced using the syntax `struct` ***tag;*** or `union` ***tag;*** which makes the type's structure tag known to the compiler, but not the structure or union's size. Usually, this error indicates that an appropriate header file has not been included.

case label must be a constant integer expression The value in a case label must be known to the compiler at compilation time and cannot depend upon runtime values. If you need to make multi-way decisions using runtime values then use a set of `if-else` statements.

'case' not inside 'switch'
You have placed a case label outside a switch statement in the body of a function. This usually happens when you edit the code and remove a trailing close brace by mistake.

cast from '*type*$_1$' to '*type*$_2$' is illegalCasting between ***type***$_1$ and ***type***$_2$ makes no sense and is illegal according to the ANSI specification. Casting, for instance, between an integer and a structure is disallowed, as is casting between structures.

cast from '*type*$_1$' to '*type*$_2$' is illegal in constant expressionCasting the pointer type ***type***$_1$ to type ***type***$_2$ is not allowed in a constant expression as the pointer value cannot be known at compile-time.

conflicting argument declarations for function '*name*'

You have declared the function ***name*** with an inconsistent prototype, such as changing the type of a parameter or not matching the number of parameters.

You have placed a `continue` statement in the main body of a function without an enclosing `for`, `do-while`, or `while-do` statement. This usually happens when you edit the code and remove a trailing close brace by mistake.

`declared parameter '`*`name`*`' is missing`

In an old-style function definition, the parameter name is declared but is missing from the function.

`'default' not inside 'switch'`

You have placed a default case label outside a switch statement in the body of a function. This usually happens when you edit the code and remove a trailing close brace by mistake.

`duplicate case label '`*`number`*`'`

You have given two or more case labels the same value — all case labels must have distinct values.

`duplicate declaration for '`*`name`*`' previously declared at `*`pos`*

You have used the name *name* to declare two objects in the same scope with identical names.

`duplicate field name '`*`name`*`' in '`*`type`*`'`

The field name *name* has already been used in the structure or union type *type*.

`empty declaration`

You have started a declaration but haven't defined an object with it. This usually occurs when declaring structure or union types as the syntax is commonly misunderstood.

`expecting an enumerator identifier`

You must use only identifiers in enumeration types, and the enumeration type must have at least one element.

`expecting an identifier`

You have not constructed an old-style parameter list correctly.

`extra 'default' cases in 'switch'`

You have supplied more than one default case label in a `switch` statement — a switch statement can have either no default case label or a single default case label.

`extraneous identifier '`*`name`*`'`

You have given more than one identifier in a declaration. This usually happens when you forget a comma.

`extraneous old-style parameter list`

You have given an old-style parameter list when declaring a pointer to a function.

`extraneous return value`

You have provided an expression to return from a void function — void functions cannot return values.

`field name expected`

You have not used an identifier to select a field name after '`.`' or '`->`'.

`field name missing`

You have not provided a field name in a structure or union declaration.

found '*type*', expected a function

You have tried to call something which is not a function.

frame exceeds *size* bytes

The size of the stack frame has exceeded ***size*** bytes which means that the application will not work at runtime.

illegal character '*char*'

The character *char* isn't valid in a C program. For example, '`$`' isn't used in C.

illegal character '\0*ooo*'

The compiler has found a non-printable character which isn't valid in a C program.

illegal expression

You have not constructed an expression correctly. This can happen for many reasons and it is impractical to list them all here.

illegal formal parameter types

You cannot specify a formal parameter as `void`, only as a pointer to `void`.

illegal initialization for '*name*'

You cannot initialize the function *name* — functions can't be initialized.

illegal initialization for parameter '*name*'

You cannot initialise parameter *name* in a formal parameter list.

illegal initialization of 'extern *name*'

You cannot initialise the value of external variables.

illegal return type; found '*type*1' expected '*type*2'

The type *type*1 of the expression returned is not compatible with the delcared return type of the function which is *type*2.

illegal return type '*type*'

You have declared a function with return type *type* which is a function or an array — a function cannot return an array nor a function.

illegal statement termination

You have not correctly terminated a statement.

illegal type '*type*'

You can use `const` and volatile only on types which are not already declared const or volatile.

illegal type '*type*' in switch expressionYou can only use simple types in switch expressions — expressions which are compatible with `int`.

illegal type '*type*[]'

You cannot declare arrays of functions, only arrays of function pointers,

illegal use of incomplete type '*type*'

The return type *type* of a function must be known before it can be called or declared as a function — that is, you cannot use incomplete types in function declarations.

illegal use of type name '*name*'

The name *name* is a typedef and cannot be used in an expression.

initializer must be constant

The value you have used to initialize a variable is not constant and is only known at run time — a constant which is computable at compile time is required.

insufficient number of arguments to '*name*'

You have not provided enough arguments to the function *name*.

integer expression must be constant

The value you have used in a bit field width, in specifying the size of an array, or defining the value of an enumeration is not constant — a constant which is computable at compile time is required.

invalid *type* field declarations

Structure and union field declarations must start with a type name.

invalid floating constant '*string*'

The string ***string*** is a invalid floating-point constant.

invalid hexadecimal constant '*string*'

The string ***string*** is a invalid hexadecimal constant.

invalid initialization type; found '*type*$_1$' expected '*type*$_2$' The type ***type***$_1$ which you have used to initialize a variable is not compatible with the type ***type***$_2$ of the variable.

invalid octal constant '*string*'

The string ***string*** is a invalid octal constant.

invalid operand of unary &; '*name*' is declared register

You cannot take the address of register variables and name is declared as a register variable.

invalid storage class '*class*' for '*type name*'

You have mis-declared the storage class for the variable ***name*** of type ***type***. For example, you cannot declare global objects `auto`, nor can you declare parameters `static` or `external`.

invalid type argument '*type*' to 'sizeof'

You cannot apply `sizeof` to an undefined type or to a function.

invalid type specification

The combination of type qualifiers and size specifiers isn't valid.

invalid use of '*keyword*'

You can't use `register` or `auto` at global level.

invalid use of 'typedef'

You can only use `typedef` to define plain types without a storage class.

left operand of . has incompatible type '*type*'

The operand to the left of the '`.`' isn't of structure type.

left operand of -> has incompatible type '*type*'

The operand to the left of the '`->`' isn't a pointer to a structure type.

lvalue required

A lvalue is required. An lvalue is an object which can be assigned to.

missing 'missing "

A string constant hasn't been closed correctly.

missing *type* tag

A structure tag is sometimes required if an undefined structure is used on its own without a typedef.

`missing { in initialization of 'type'`
Types with nested structures or unions must be initialised correctly with structures delimited with '`{`' and '`}`'.

`missing array size`
When declaring an array without an initialiser, you must give explicit array sizes.

`missing identifier`
Your declaration is missing an identifier which defines what is being declared.

`missing label in goto`
An identifier is required immediately after `goto`.

`missing name for parameter number to function 'name'`
Only function ptototypes can have anonymous parameters; for ANSI-style function declarations all parameters must be given names,

`missing parameter type`
An ANSI-style function declaration requires that all parameters are typed in the function prototype.

`operand of unary operator has illegal type 'type'`
The operand's type isn't compatible with the unary operator *operator*.

`operands of operator have illegal types 'type1' and 'type2'` The types of the left and right operands to the binary operator ***operator*** are not allowed.

`overflow in value for enumeration constant 'name'`
When declaring ***name*** as an enumation constant, the integer value of that constant exceeds the maximum integer value.

`redeclaration of 'name' redeclaration of 'name' previously declared at position`
The identifier ***name*** has already been used in this scope for another purpose and cannot be used again.

`redefinition of 'name' previously defined at position`
You have redefined the initialisation of the identifier ***name*** — only one definition of the identifier's value is allowed.

`redefinition of label 'name' previously defined at position`
You have redefined the label ***name*** with the same name in the function. Labels are global to the function, not local to a block.

`size of 'type' exceeds size bytes`
The size of the type ***type*** is greater than ***size*** bytes. The compiler cannot construct data items larger than ***size*** bytes for this processor.

`size of 'type[]' exceeds size bytes`
The size of the array of ***type*** is greater than ***size*** bytes. The compiler cannot construct data items larger than ***size*** bytes for this processor.

`'sizeof' applied to a bit field`
You cannot use `sizeof` with a bit field as `sizeof` returns the number of bytes used for an object, whereas a bit field is measured in bits.

too many arguments to '*name*'

You have provided too many arguments to the function *name*.

too many errors

The compiler has stopped compilation because too many errors have been found in your program. Correct the errors and then recompile.

too many initializers

You have provided more initializers for an array or a structure than the compiler expected. Check the bracketing for nested structures.

type error in argument *n* to *name*; '*type*' is illegal

The ***n***th actual argument to the function *name* is of type *type* and is not compatible with the ***n***th formal argument given in the prototype for *name*. You should check that the formal and actual parameters on a function call match.

type error in argument *n* to *name*; found '*type*1' expected '*type*2' The ***n***th actual argument to the function *name* is of type *type*1 and is not compatible with the ***n***th formal argument given of type ***type***2 in the prototype for *name*. You should check that the formal and actual parameters on a function call match.

undeclared identifier '*name*'

You have used the identifier *name* but it has not been previously declared.

undefined label '*name*'

You have used the label ***name*** in a goto statement but no label in the current function has been defined with that name.

undefined size for '*type name*'

You have declared the variable ***name*** using the type ***type***, but the size of type is not yet known. This occurs when you define a union or structure with a tag but do not define the contents of the structure and then use the tag to define a variable.

undefined size for field '*type name*'

You have declared the field ***name*** using the type ***type***, but the size of type is not yet known. This occurs when you define a union or structure with a tag but do not define the contents of the structure and then use the tag to define a field.

undefined size for parameter '*type name*'

You have declared the parameter ***name*** using the type ***type***, but the size of type is not yet known. This occurs when you define a union or structure with a tag but do not define the contents of the structure and then use the tag to define a parameter.

undefined static '*type name*'

You have declared the static function ***name*** which returns type ***type*** in a prototype, but have not defined the body of the function.

unknown enumeration '*name*'

You have not defined the enumeration tag name but have used it to defined an object.

unknown field '*name*' of '*type*'

You have tried to access the field ***name*** of the structure or union type ***type***, but a field of that name is not declared within that structure.

unknown size for type '*type*'

You have tried to use an operator where the size of the type *type* must be known to the

compiler.

unrecognized declaration

The compiler can't recognise the declaration syntax you have used. This is usually caused by misplacing a comma in a declarator.

unrecognized statement

The compiler can't recognise the start of a statement. A statement must start with one of the statement keywords or must be an expression. This is usually caused by misplacing a semicolon.

user type check error: found '*type*2' expected '*type*2'

The operand to the `__typechk` intrinsic function is incorrect. This occurs when you provide a parameter to a the macro function which checks its expected parameters using `__typechk`. You should check the types of parameters you pass to the macro routine.

End of document

