PUMPKIN™
REAL-TIME SOFTWARE

AN-31
Application Note

750 Naples Street • San Francisco, CA 94112 • (415) 584-6360 • http://www.pumpkininc.com

Building a Salvo Application with Keil's CARM C Compiler and µVision IDE

Introduction

This Application Note explains how to use Keil's (http://www.keil.com/) CARM compiler and µVision IDE to create a multitasking Salvo application for microcontrollers with embedded ARM7TDMI cores.

We will show you how to build the Salvo application contained in `\salvo\ex\ex1\main.c` for a Philips (http://www.philips.com/) LPC2129 using the Keil tools.

For more information on how to write a Salvo application, please see the *Salvo User Manual*.

Before You Begin

If you have not already done so, install the CARM and µVision tools. With the µVision IDE you will be able to run and debug this application in the simulator or on real hardware (if available).

Related Documents

The following Salvo documents should be used in conjunction with this manual when building Salvo applications with Keil's CARM compiler and µVision IDE:

Salvo User Manual
Salvo Compiler Reference Manual RM-KCARM

created by Andrew E. Kalman on Oct 16, 2004 updated on Oct 17, 2004

Creating and Configuring a New Project

Create a new µVision project using Project → New Project. In the Create New Project window, navigate to your working directory (in this case we've chosen `c:\temp`) and enter a name for the project (we'll use `myex1`) in the File Name field:

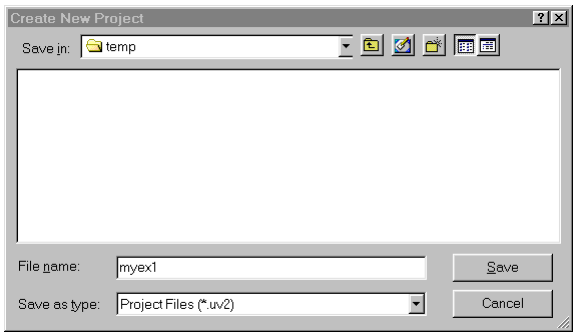

Figure 1: Creating the New Project

Click on Save to continue. The Select Devices for Target 'Target 1' window appears. Under the CPU tab select and expand Philips:

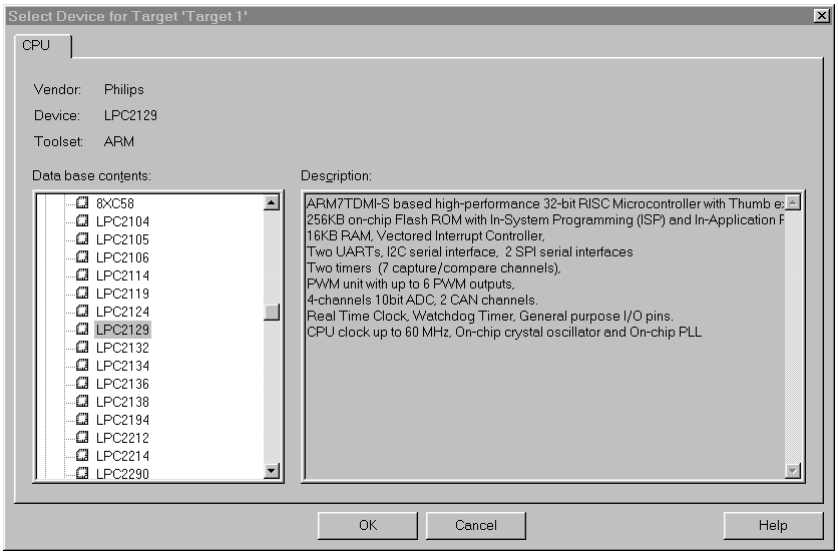

Figure 2: µVision Device Selection Window with Philips LPC2129 Selected

Select LPC2129 and click on OK to continue. You'll be prompted to copy and add target-specific startup code to the project. Select Yes to continue:

Figure 3: Confirming Addition of Startup Code to Project

The Keil file `Startup.s` will be added to the project.

Preprocessor Options

Now let's setup the project's options for Salvo's pathnames, etc. Choose Project → Options for Target 'Target 1' → C and define any symbols you may need for your project in the Preprocessor Symbols → Define area.[1] In the Include Paths, add `\salvo\inc`:

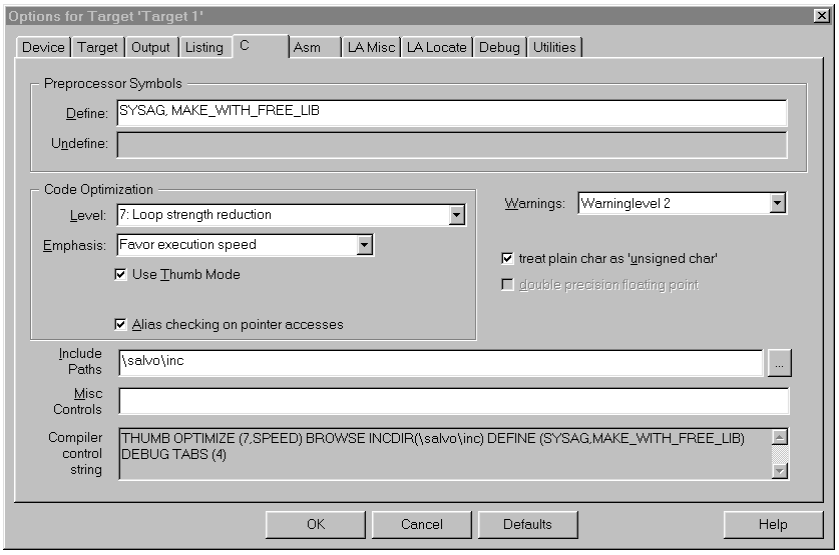

Figure 4: CARM C Options for Target

Note This project options screen is also used to select Thumb mode or ARM mode. The CARM default is Thumb mode.

Click on OK to continue.

Groups

In order to manage your project effectively, we recommend that you create a set of groups for your project. They are:

Listings
Salvo Configuration File
Salvo Help Files
Salvo Libraries
Salvo Sources
Sources

For each group, choose Project → Components, Environment and Books, and under Project Components → Groups add and (re-)order the new group names[2], and select OK:

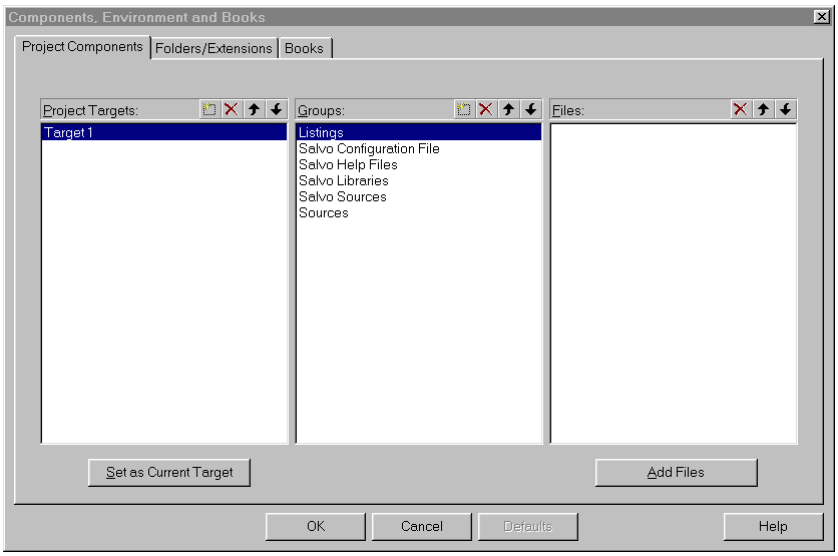

Figure 5: Creating a Group

When finished, your project window should look like this:

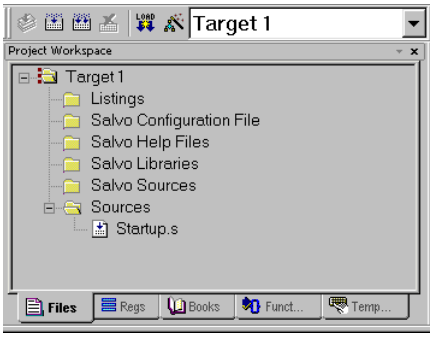

Figure 6: Project Window with Groups

Compiler Selection

Lastly, you'll need to configure this project for use with Keil's CARM compiler. Choose Project → Components, Environment and Books, and under Folders/Extensions → Select ARM Development Tools select Use Keil ARM Tools. Additionally, under Development Tools Folders, select Use Settings from TOOLS.INI:

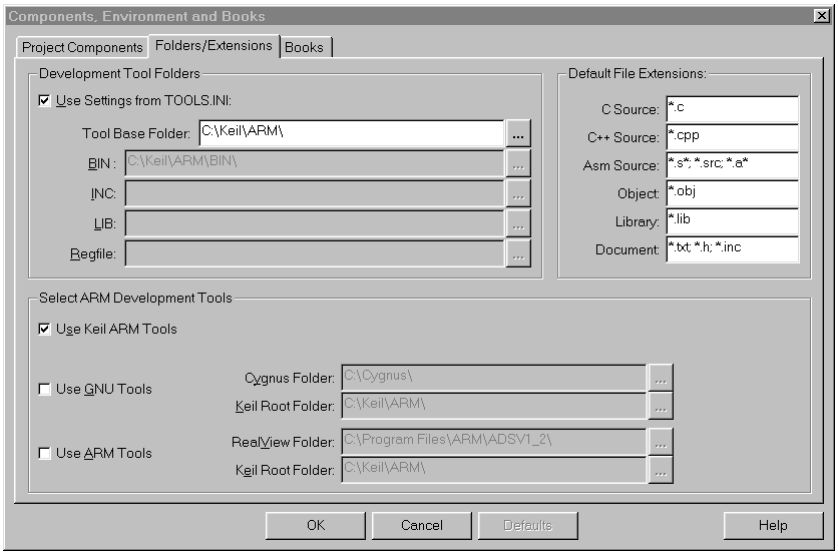

Figure 7: Selecting the CARM Compiler

Click on OK to finish configuring your project.

Adding your Source File(s) to the Project

Now it's time to add files to your project. Choose Project → Components, Environment and Books, and under Project Components → Groups select Sources. Click on Add Files, navigate to your project's directory, select your `main.c` and Add, then Close. Your Project Files window should look like this:

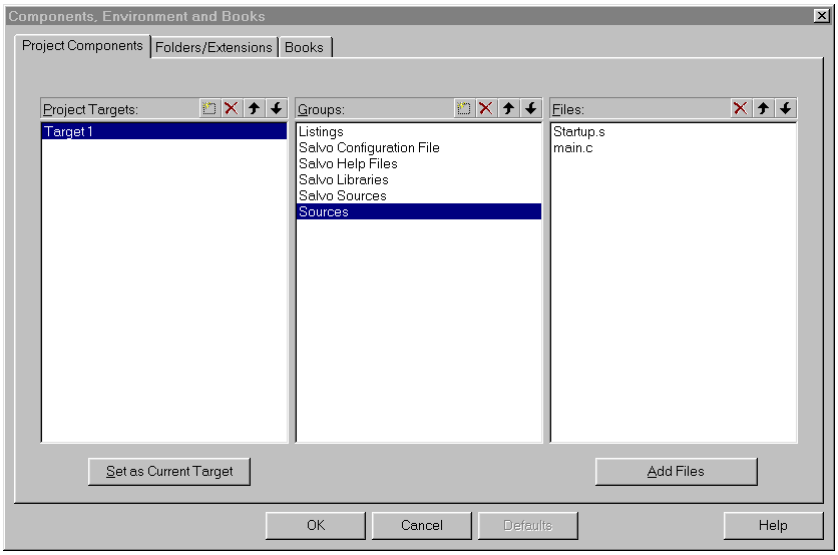

Figure 8: Adding Source Files to the Project

When finished, select OK, and your project window should look like this:

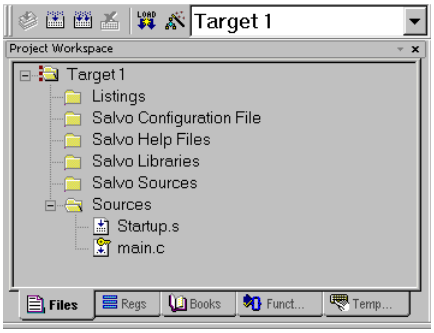

Figure 9: Project Window with Project-Specific Source Files

Adding Salvo-specific Files to the Project

Now it's time to add the Salvo files your project needs. Salvo applications can be built by linking to precompiled Salvo libraries, or with the Salvo source code files as nodes in your project.

Adding a Salvo Library

For a *library build* – e.g. what you would do when evaluating Salvo via Salvo Lite – a fully-featured Thumb-mode little-endian Salvo freeware library for the Philips LPC2129 (and all microcontrollers based on the ARM7TDMI and related cores) is `sfkcarm4lt-a.lib`.[3] Choose Project → Components, Environment and Books, and under Project Components → Groups select Salvo Libraries. Click on Add Files, navigate to the `\salvo\lib\kcarm` directory, select `sfkcarm4lt-a.lib` and Add, then Close. Your Project Files window should look like this:

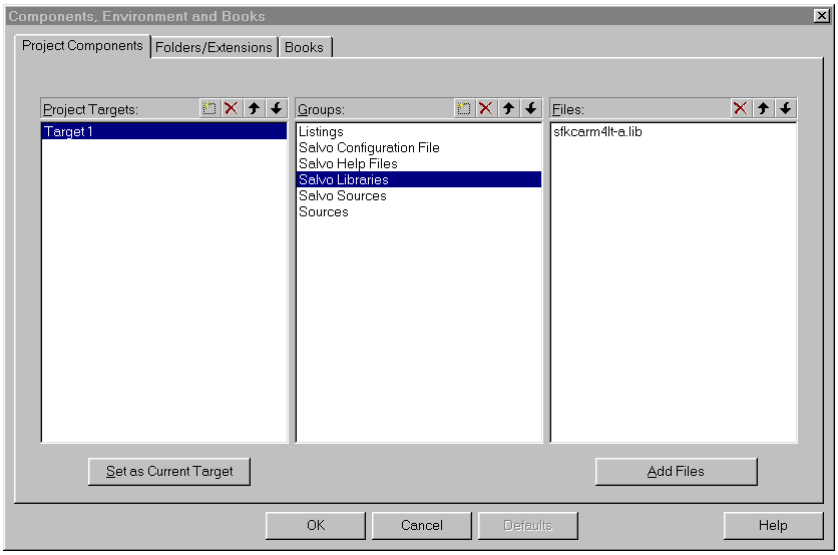

Figure 10: Adding Salvo Libraries to the Project

When finished, select OK, and your project window should look like this:

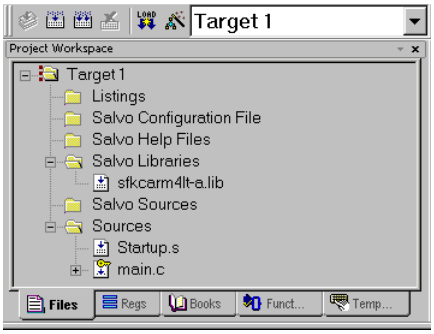

Figure 11: Project Window with Salvo Libraries

You can find more information on Salvo libraries in the *Salvo User Manual* and in the *Salvo Compiler Reference Manual RM-KCARM.*.

Adding Salvo's mem.c

Salvo library builds also require Salvo's `mem.c` source file as part of each project. Choose Project → Components, Environment and Books, and under Project Components → Groups select Salvo Sources. Click on Add Files, navigate to the `\salvo\src` directory, select `mem.c` and Add, then Close. Your Project Files window should look like this:

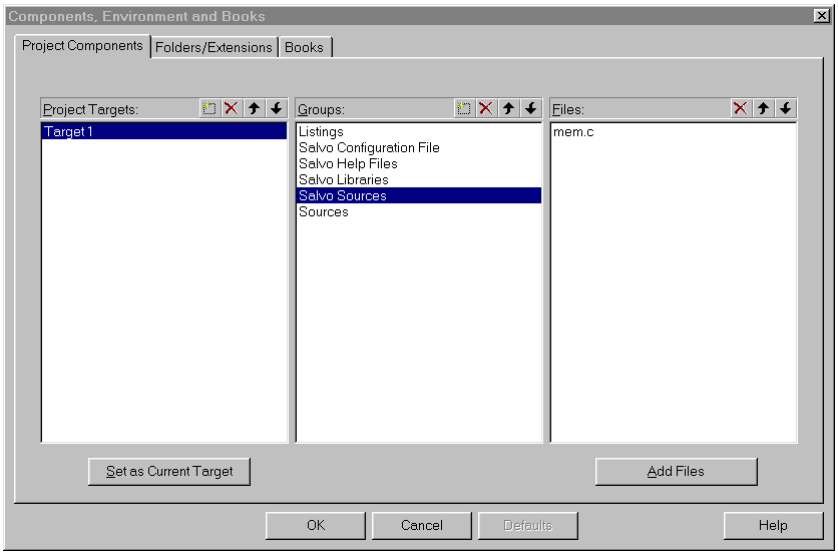

Figure 12: Adding Salvo's mem.c to the Project

When finished, select OK, and your project window should look like this:

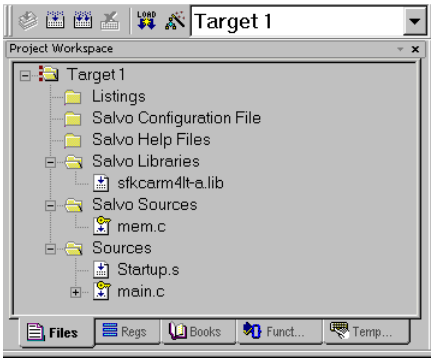

Figure 13: Project Window with Salvo Libraries

The salvocfg.h Header File

You will also need a `salvocfg.h` file for this project. To use the library selected in Figure 10, your `salvocfg.h` should contain only:

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OSA
#define OSEVENTS                1
#define OSEVENT_FLAGS           0
#define OSMESSAGE_QUEUES        0
#define OSTASKS                 3
```

Listing 1: Example salvocfg.h for a Salvo Lite Library Build

Note The settings above are for this particular example project. The settings for your projects will vary depending on which libraries you use, how many tasks and events are in your application, etc.

For your convenience, you'll want your project's `salvocfg.h` to be easily accessible. Choose Project → Components, Environment and Books, and under Project Components → Groups select Salvo Configuration File. Click on Add Files, navigate to your project's directory, select `salvocfg.h` and Add, then Close. Your Project Files window should look like this:

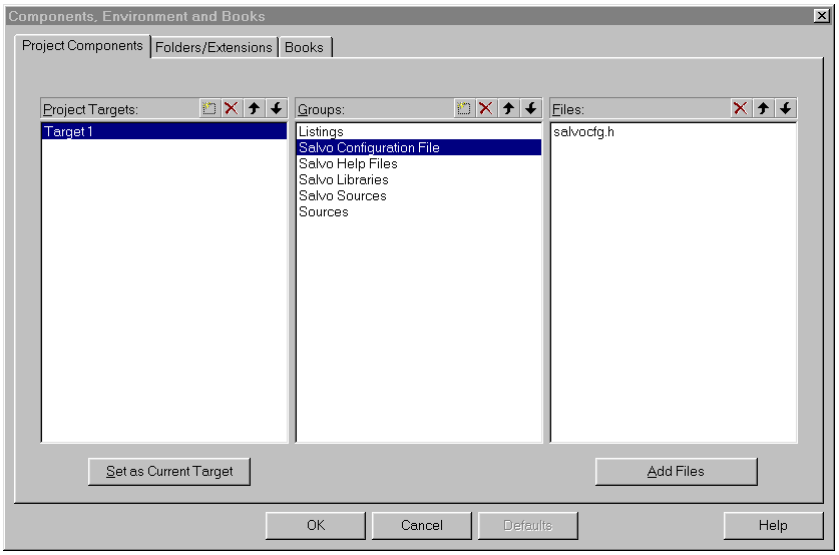

Figure 14: Adding the Configuration File to the Project

When finished, select OK, and your project window should look like this:

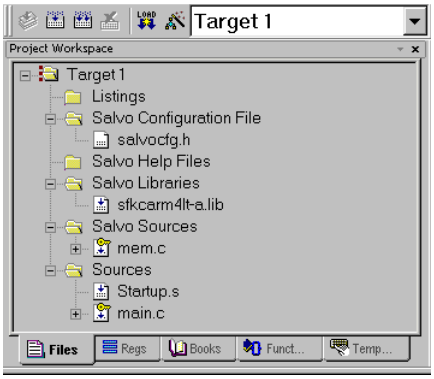

Figure 15: Project Window for a Library Build

Tip The advantage of placing the various project files in the groups shown above is that you can quickly navigate to them and open them for editing, etc.

Proceed to *Building the Project*, below.

Adding Salvo Source Files

If you have a Salvo Pro distribution, you can do a *source code build* instead of a library build. The application in `\salvo\ex\ex1\main.c` contains calls to the following Salvo user services:

```
OS_Delay()              OSInit()
OS_WaitBinSem()         OSSignalBinSem()
OSCreateBinSem()        OSSched()
OSCreateTask()          OSTimer()
OSEi()
```

You must add the Salvo source files that contain these user services, as well as those that contain internal Salvo services, to your project. The *Reference* chapter of the *Salvo User Manual* lists the source file for each user service. Internal services are in other Salvo source files. For this project, the complete list is:

```
binsem.c                mem.c
delay.c                 portkcarm.s
event.c                 qins.c
idle.c                  sched.c
init.c                  timer.c
inittask.c              wdtctrl.c
intctrl.c
```

To add these files to your project, choose Project → Components, Environment and Books, and under Project Components → Groups select Salvo Sources. Click on Add Files, navigate to the `\salvo\src` directory, select the files listed above and Add, then Close. Your Project Files window should look like this:

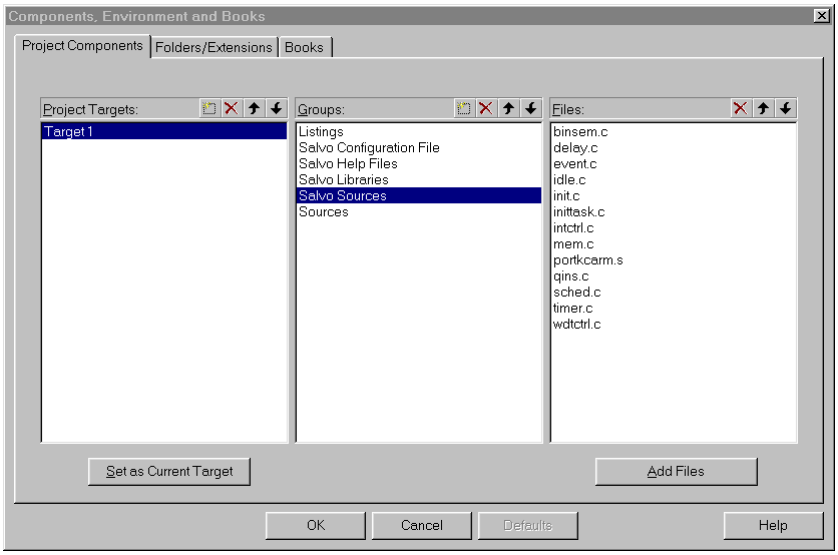

Figure 16: Adding Salvo Source Files to the Project

When finished, select OK, and your project window should look like this:

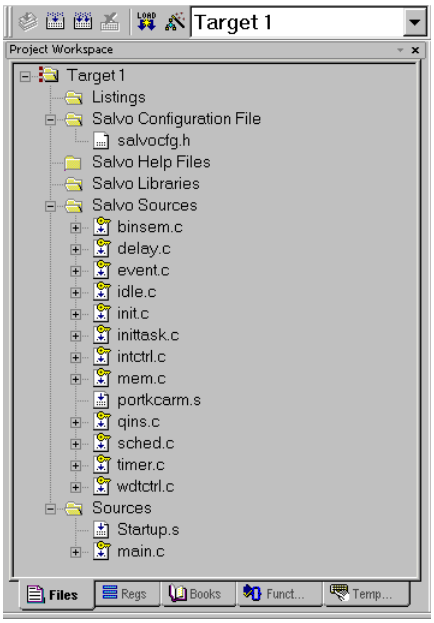

Figure 17: Project Window for a Source-Code Build

The salvocfg.h Header File

You will also need a `salvocfg.h` file for this project. Configuration files for source code builds are quite different from those for library builds (see Listing 1, above). For a source code build, the `salvocfg.h` for this project contains only:

```
#define OSENABLE_IDLING_HOOK         TRUE
#define OSENABLE_BINARY_SEMAPHORES   TRUE
#define OSEVENTS                     1
#define OSEVENT_FLAGS                0
#define OSMESSAGE_QUEUES             0
#define OSTASKS                      3
```

Listing 2: salvocfg.h for a Source Code Build

Note The settings above are for this particular example project. The settings for your projects will vary depending on which configurations you use, how many tasks and events are in your application, etc.

Building the Project

For a successful compile, your project must also include a header file (e.g. `#include <LPC21xx.h>`) for the particular chip you are using. Normally, this is included in each of your source files (e.g. `main.c`), or in a header file that's included in each of your source files (e.g. `main.h`).

With everything in place, you can now build the project using Project → Build Target or Project → Rebuild all target files. The build results can be seen in the Build window:

```
Build target 'Target 1'
compiling mem.c...
assembling Startup.s...
compiling main.c...
linking...
Program Size: data=1259 const=24 code=2204
"myex1" - 0 Error(s), 0 Warning(s).
```

Listing 3: Salvo Lite Library Build Results

This example uses a total of 1259 bytes of RAM in the `data` space (see Note below), 24 bytes in the `const` space, and 2204 bytes of ROM in the `code` space.

Note The `data` space total shown in Listing 3 includes all stacks in the application. In this example, the default `Startup.s` file has allocated a total of 0x490 (1168) bytes to the Undefined, Supervisor, Abort, Fast Interrupt, Interrupt and User/System Mode stacks. Therefore in this example, the project-specific RAM size

(including the RAM used by Salvo) is 1259-1168 = 91 bytes of RAM.

Note The µVision projects supplied in the Salvo for ARM distributions contain additional help files in each project's Salvo Help Files group.

Viewing Salvo Documentation

Salvo documentation is available directly from within µVision's project window. Simply select the Books tab and expand the Tool User's Guide group:

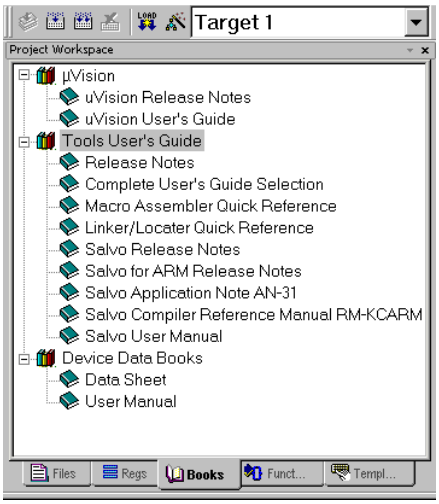

Figure 18: Salvo Documentation in Project Window's Books Tab

Testing the Application

You can test and debug this application using the µVision debugger and its simulator, a Flash programming utility, or the optional ULINK JTAG interface

Simulator

Choose Project → Options for Target 'Target 1' → Debug and select Use Simulator.

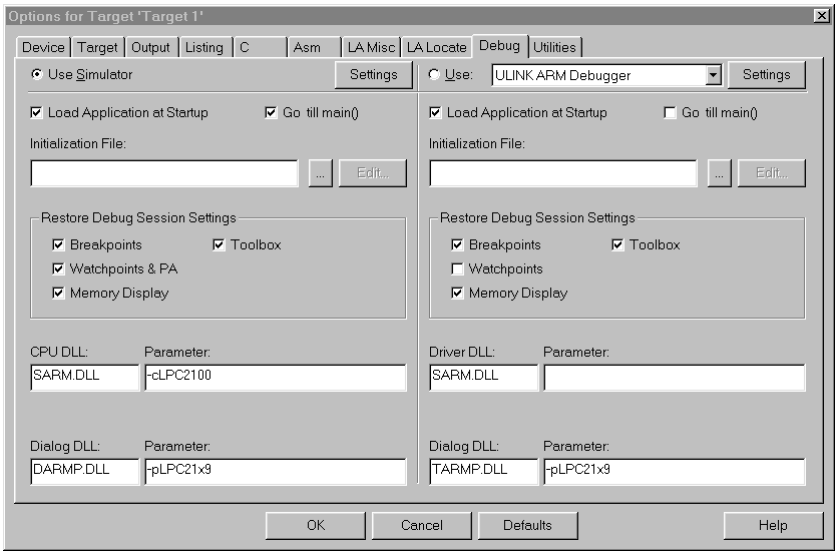

Figure 19: Selecting the Simulator

Select OK to continue. Choose Debug → Start/Stop Debug Session → Go and the Salvo application will begin executing in the simulator. When code execution is interrupted via Stop Running, the debugger will locate the PC at the current instruction. If that instruction is in user or Salvo source code, the debugger will stop within C source code. Otherwise it will stop and display disassembly.

Within the simulator you can view peripherals and registers, watch variables of interest, single-step in C or assembly, set breakpoints, etc:

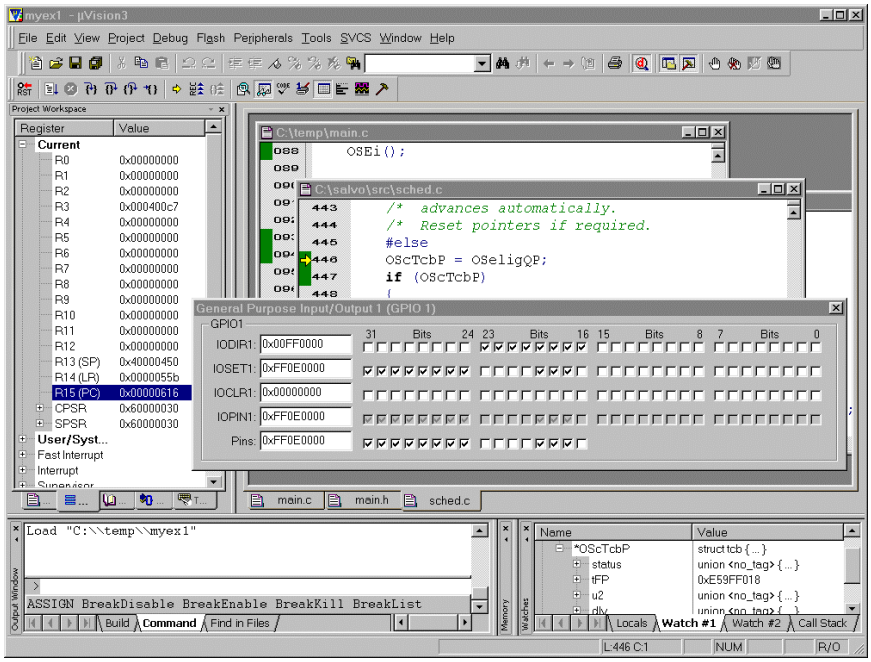

Figure 20: Single-stepping in the Simulator

Note µVision supports debugging at the source code level. Only applications built from the Salvo source code enable you to step through Salvo services (e.g. `OSCreateBinSem()`) at the source code level. Regardless of how you build your Salvo application, you can always step through your own C and assembly code in the µVision debugger.

Flash Download

Application code can be downloaded to target hardware using external tools (e.g. the Philips LPC2000 Flash Utility) or via Keil's ULINK ARM debugger.

Flash Utilities

For use with Flash utilities, you'll need to force µVision to generate a HEX file. Choose Project → Options for Target 'Target 1' → Output and select Create HEX File:

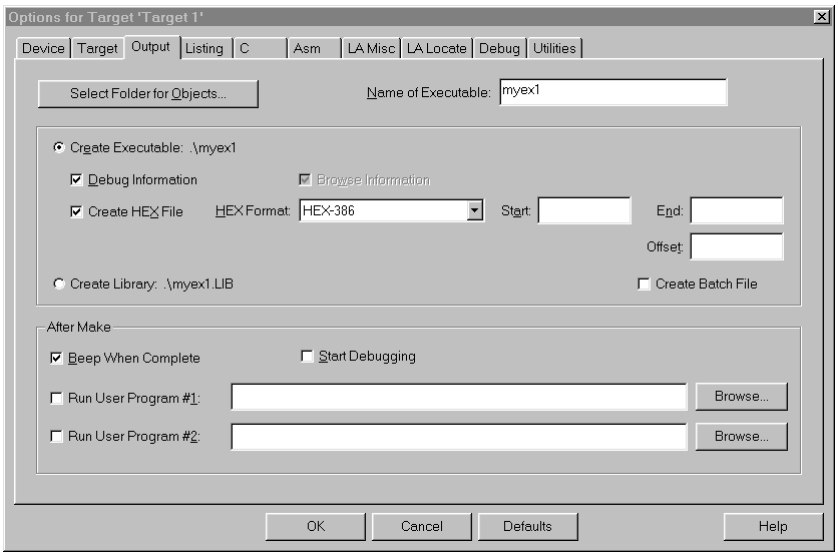

Figure 21: Generating a HEX file

Select OK to continue. Rebuild the application. Launch the Flash utility, and navigate to your project's `.hex` output file to select the file for downloading:

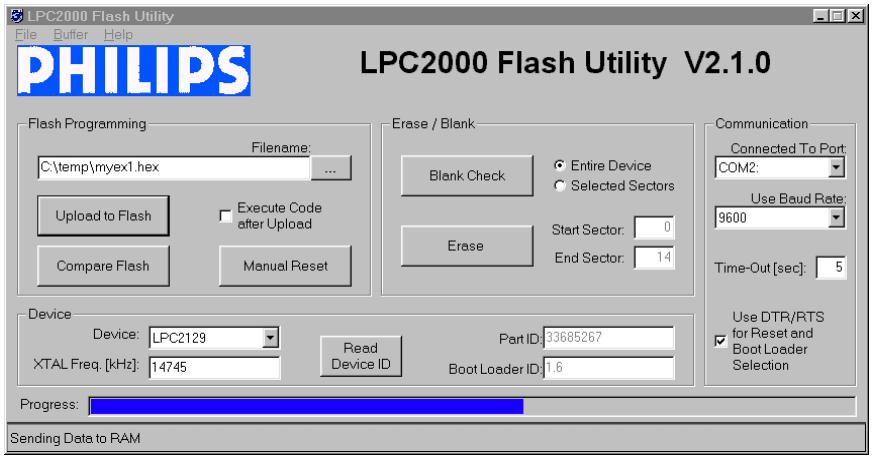

Figure 22: Downloading the HEX file via a Flash Utility

Download the `.hex` file to your target. You'll probably have to manually reset the target to begin execution.

You can also launch the flash utility directly from within µVision. Choose Project → Options for Target 'Target 1' → Utilities and select Use External Tool for Flash Programming.

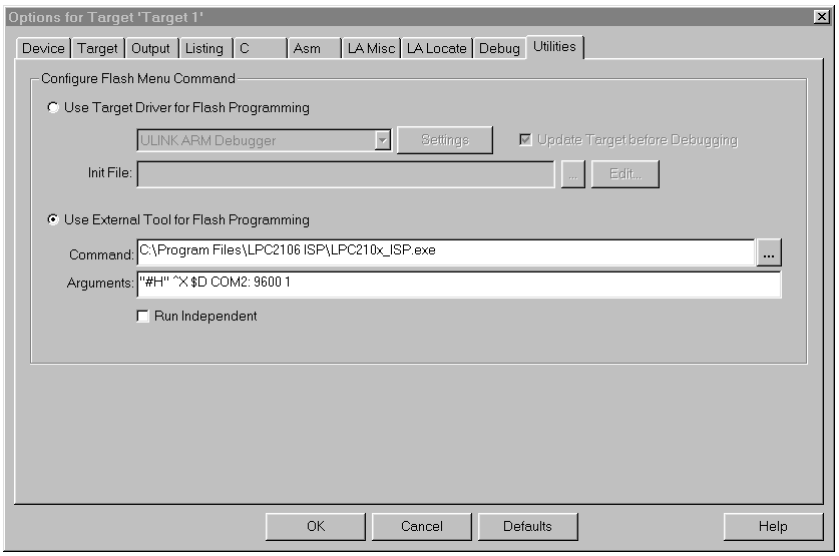

Figure 23: Integrating a Flash Programming Utility into µVision

Select OK to continue. With the Command line properly configured, you can download via the Flash utility simply by choosing Flash → Download.

ULINK

Keil's ULINK ARM debugger provides the ability to debug on real hardware over a JTAG port. Choose Project → Options for Target 'Target 1' → Utilities and select ULINK ARM Debugger under the Use Target Driver for Flash Programming pull-down. You'll probably want to select Update Target before Debugging to streamline your debugging sessions.

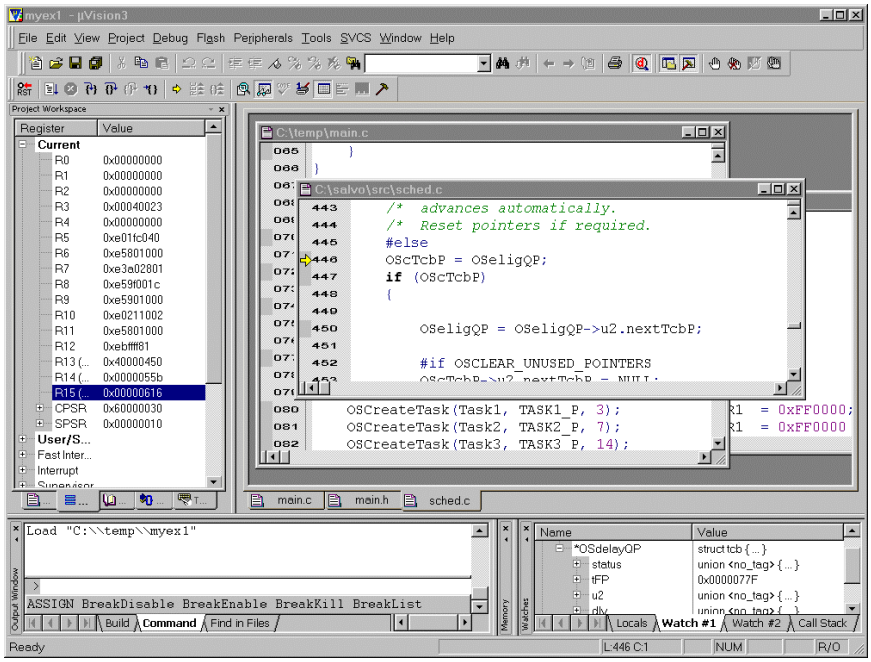

Figure 24: Single-stepping with ULINK

Troubleshooting

Compiler Error: Can't Open File …

Failure to add the `\salvo\inc` path to the C compiler's include paths will cause the following compiler error:

```
Build target 'Target 1'
compiling mem.c...
C:\salvo\src\mem.c(30): error C318: can't open file
'salvo.h'
```

Listing 4: Compiler Error due to Missing Salvo Include Path

See *Preprocessor Options* above for how to add the Salvo include path to the project's C compiler options.

Failure to have a `salvocfg.h` configuration file in the project's directory will cause the following compiler error:

```
assembling Startup.s...
compiling main.c...
\SALVO\INC\SALVO.H(98): error C318: can't open file
'salvocfg.h'
\SALVO\INC\SALVO.H(130): error C320: salvo.h:
salvocfg.h: OSTASKS undefined -- aborting.
```

Listing 5: Compiler Error due to Missing salvocfg.h Configuration File

See The salvocfg.h Header Fileabove for how to add the Salvo configuration file to the project.

Linker Error: Unresolved Externals …

Missing mem.c

Failure to add `\salvo\src\mem.c` to the project will cause the following linker error:

```
Build target 'Target 1'
assembling Startup.s...
compiling main.c...
linking...
*** WARNING L23: UNRESOLVED EXTERNAL SYMBOLS
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSecbArea
    ADDRESS: 000001A4H
[SNIP]
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSframeP
    ADDRESS: 000006F0H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OScTcbP
    ADDRESS: 000006FEH
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OScTcbP
    ADDRESS: 0000072CH
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSdelayQP
    ADDRESS: 00000774H
Program Size: data=1168 const=16 code=2088
Target not created
```

Listing 6: Linker Error due to Missing Salvo mem.c

Note The unresolved externals in Listing 6 reference Salvo's *global variables*. Their names begin with a lower-case letter and are prefixed by `OS`, e.g. `OScTcbP`.

See *The salvocfg.h Header File* above for how to add Salvo's `mem.c` file to your project.

Missing Library

Failure to add a Salvo library (in library builds) or the appropriate Salvo source file(s) (in source-code builds) to the project will cause the following linker error:

```
Build target 'Target 1'
compiling mem.c...
assembling Startup.s...
compiling main.c...
linking...
*** WARNING L23: UNRESOLVED EXTERNAL SYMBOLS
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSDelay?T
    ADDRESS: 00000190H
[SNIP]
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSCreateBinSem?T
    ADDRESS: 000002B4H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSEnableInts?T
    ADDRESS: 000002BEH
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSSched?T
    ADDRESS: 000002C8H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSTimer?A
    ADDRESS: 00000338H
Program Size: data=1259 const=24 code=864
Target not created
```

Listing 7: Linker Error due to Missing Salvo Library

Note The unresolved externals in Listing 7 reference *services* in Salvo's *API*. Their names begin with an upper-case letter and are prefixed by `OS`, e.g. `OSTimer()`. The `?T` after the function name indicates that the unresolved external is a Thumb-mode function.

See *Adding a Salvo Library* and *Adding Salvo Source Files*, above, for how to ensure that the linker can find the Salvo services referenced in the application.

Thumb-vs-ARM Mode Mismatch

Failure to ensure that all components in a project are built in the same CPU mode (Thumb or ARM) will cause the following linker error:

```
Build target 'Target 1'
compiling mem.c...
assembling Startup.s...
compiling main.c...
linking...
*** WARNING L23: UNRESOLVED EXTERNAL SYMBOLS
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSDelay?A
    ADDRESS: 000001D0H
[SNIP]
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSCreateBinSem?A
    ADDRESS: 000003B0H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSEnableInts?A
    ADDRESS: 000003C0H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSSched?A
    ADDRESS: 000003D0H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSTimer?A
    ADDRESS: 0000042CH
Program Size: data=1259 const=24 code=1108
Target not created
```

Listing 8: Linker Error due to Missing Salvo Library

In Listing 8, the project was built in ARM mode, but the Salvo library used only supports Thumb mode. As a result of this mismatch, the linker looked for but could not find Salvo services in ARM mode (hence the `?A`'s above).

Salvo libraries for Keil's CARM C compiler are available in pure Thumb, pure ARM, and mixed-mode versions. You must ensure that when using pure Thumb or pure ARM versions, all of the files in the project are compiled using the same mode.

See *Preprocessor Options* above for how to set Thumb or ARM mode. See the *Salvo Compiler Reference Manual RM-KCARM* for more information on Salvo libraries for Keil's CARM C compiler.

Linker Error: Data Types Different …

In a library build, adding a library to a project that does not conform to the configuration options in the project's `salvocfg.h` will cause the following linker error:

```
Build target 'Target 1'
compiling mem.c...
assembling Startup.s...
compiling main.c...
linking...
*** WARNING L25: DATA TYPES DIFFERENT
    SYMBOL:  OSeligQP
    MODULE:  C:\salvo\lib\kcarm\sfkcarm4lt-m.lib (init)
   DEFINED:  .\mem.obj (mem)
*** WARNING L25: DATA TYPES DIFFERENT
    SYMBOL:  OScTcbP
    MODULE:  C:\salvo\lib\kcarm\sfkcarm4lt-m.lib (init)
   DEFINED:  .\mem.obj (mem)
[SNIP]
*** WARNING L23: UNRESOLVED EXTERNAL SYMBOLS
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSDelay?T
    ADDRESS: 00000190H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSWaitBinSem?T
    ADDRESS: 000001B6H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSDelay?T
    ADDRESS: 000001EAH
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSSignalBinSem?T
    ADDRESS: 00000204H
*** ERROR L128: REFERENCE MADE TO UNRESOLVED EXTERNAL
    SYMBOL:  OSCreateBinSem?T
    ADDRESS: 0000028CH
Program Size: data=1259 const=24 code=1268
Target not created
```

Listing 9: Linker Error due to Wrong Salvo Library

In Listing 9, the project called for the Salvo library `sfkcarm4lt-a.lib`, but the project contained the library `sfkcarm4lt-m.lib`. As a result of this mismatch, various data types are different, and some required Salvo services were not found.

See the *Salvo Compiler Reference Manual RM-KCARM* for more information on Salvo libraries for Keil's CARM C compiler.

Application Crashes

After Changing Processor Type

Remember to `#include` the appropriate header file for your ARM7 microcontroller (see *Building the Project*, above). A common cause for such crashes is a difference in interrupt vector locations or definitions between two members of a processor family. Mainline code may work correctly, but the application will crash if interrupt vectors are not in the right locations.

When Interrupts are Enabled

Salvo libraries and `\salvo\src\intctrl.c` contain dummy versions of the user control functions `OSDisableInts()`, `OSEnableInts()`, `OSRestoreInts()` and `OSSaveInts()`. If your application uses interrupts, you must create your own user control functions to ensure that Salvo's critical sections are protected from interrupts. See the *Salvo Compiler Reference Manual RM-KCARM* for more information on interrupt control.

Also ensure that (if present) you've defined the default interrupt vector for your target's Vectored Interrupt Controller (VIC). Failure to define this vector as a dummy ISR may lead to instability due to spurious interrupts.

When the Watchdog is Enabled

Salvo libraries and `\salvo\src\wdtctrl.c` contain a dummy versions of the user control function `OSClrWdt()`. If your application enables the watchdog and you want Salvo to clear the watchdog, you must create your own user control function to ensure that the watchdog timer is cleared. See the *Salvo Compiler Reference Manual RM-KCARM* for more information on the watchdog timer.

Example Projects

Example projects for Keil's CARM C compiler can be found in the `\salvo\tut\tu1-6\sysag` directories. The include path for each of these projects includes `\salvo\tut\tu1\sysag` and each project defines the `SYSAG` ymbol.

Complete Salvo Lite library-build projects are contained in the project files `\salvo\tut\tu1-6\sysag\tu1-6lite.*`. These projects also define the `MAKE_WITH_FREE_LIB` symbol.

Complete Salvo LE library-build projects are contained in the project files `\salvo\tut\tu1-6\sysag\tu1-6le.*`. These projects also define the `MAKE_WITH_STD_LIB` symbol.

Complete Salvo Pro source-code-build projects are contained in the project files `\salvo\tut\tu1-6\sysag\tu1-6pro.*`. These projects also define the `MAKE_WITH_SOURCE` symbol.

---

1 The Salvo Lite project ex1lite, upon which this example is based, supports a wide variety of targets and compilers. For use with µVision and the CARM compiler, it requires the `SYSAG` and `MAKE_WITH_FREE_LIB` defined symbols. When you write your own projects using your own source code, you may not require any symbols.

2 Groups can be renamed in this window.

3 This Salvo Lite library contains all of Salvo's basic functionality. The corresponding Salvo LE and Pro library is slkcarm4lt-a.lib.