OS-9 for SideARM Board Guide

Version 3.2

www.radisys.com

World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision A
December 2001

Copyright and publication information

This manual reflects version 3.2 of Enhanced OS-9 for StrongARM.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

December 2001

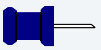
Table of Contents

Chapter 1: Installing and Configuring OS-9

This chapter describes installing and configuring OS-9 on the INTEL SA-1100 Microprocessor Reference Platform (SideARM). It includes the following sections:

- **Requirements and Compatibility**
- **Target Hardware Setup**
- **Connecting the Target to the Host**
- **Building the OS-9 ROM Image**
- **Creating a Startup File**
- **Optional Procedures**

Requirements and Compatibility



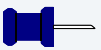
Note

Before you begin, install the ***Enhanced OS-9 for StrongARM*** CD-ROM on your host PC.

Host Hardware Requirements (PC Compatible)

Your host PC should have the following:

- Windows 95, 98, ME, 200, or Windows NT
- A minimum of 32MB of free disk space (an additional 235MB of free disk space is required to run PersonalJava Solution for OS-9)
- An Ethernet network card
- A PCMCIA card reader/writer
- At least 16MB of RAM



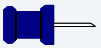
Note

If you are a PersonalJava Solution licensee and you plan to use the Java JCC to pre-load your Java classes, you may need as much as 64MB of RAM. Refer to the document ***Using JavaCodeCompact for OS-9*** for a complete discussion of using the JCC.

Host Software Requirements (PC Compatible)

Your host PC should have a terminal emulation program (such as `Hyperterminal`, which comes with Microsoft Windows.

Target Hardware Requirements



Note

Please refer to the ***SA-1100 Microprocessor Evaluation Platform User's Guide*** for information on hardware preparation and installation, operating instructions, and functional descriptions prior to installing and configuring OS-9.

Your reference board requires the following hardware:

- Enclosure or chassis with power supply
- A RS-232 null modem serial cable
- P/S2 keyboard, P/S2 mouse (for use with MAUI), and VGA monitor

Java Hardware Requirements

Your reference board must have the following to run PersonalJava Solution for OS-9:

- 16MB of RAM
- 4MB of FLASH (Boot)
- VGA monitor
- You have followed the start-up procedure for your SA-1100 evaluation board.

Target Hardware Setup

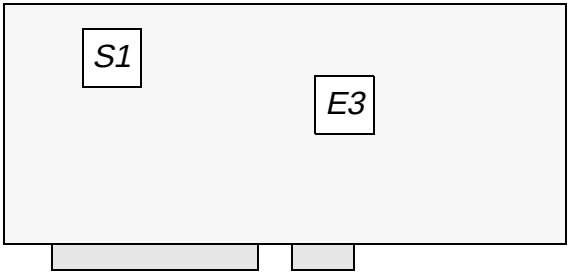
Configure Board Switch Settings

Switch S1, position 3, must be in the UP position for correct booting.

Installing the Flash Device

The first stage in configuring your reference board is to install the pre-loaded FLASH device included in your Enhanced OS-9 for StrongARM package. This device includes a coreboot system that has been pre-configured to get your board up and running quickly. Install the FLASH device in socket E3.

Figure 1-1 Installing the Flash Devices



Note

If you need to reprogram the flash device or create a new flash device, see the **Creating a new OS-9 Coreboot Image in Flash Memory** section.

Configuring the ATA Card

You can use your ATA card to validate that your reference board is operational without requiring the connection to the host machine:

To configure the ATA card:

Step 1. From a DOS prompt on the host machine, navigate to the following directory:

```
MWOS\OS9000\ARMV4\PORTS\SIDEARM\BOOTS\SYSTEMS\PORTBOOT
```

and run `os9make`.

Step 2. On the host machine, copy the files located in the following directory:

```
MWOS\OS9000\ARMV4\PORTS\SIDEARM\BOOTS\SYSTEMS\PORTBOOT\os9kboot
```

into the root directory to the ATA card

Step 3. Install the card in socket 1 (top) on the reference board

Step 4. Turn on the reference board.

Connecting the Target to the Host

Connect an RS-232 null modem cable from the reference board to the serial port of a Windows 95, Windows 98, or Windows NT system.

Step 1. Connect the serial cable to the J11 connector on the reference board. The J11 connector is serial port 1.

Step 2. Connect the other end of the serial cable to the Host PC.

Step 3. On the Windows desktop, click on the `Start -> Programs -> Accessories -> Hyperterminal`.

Step 4. Once Hyperterminal is open, enter a name for your session.

Step 5. Select an icon for the new Hyperterminal session. A new icon is created with the name of your session associated with it. Click `OK`.

Step 6. In the **Phone Number** dialog, go to the **Connect Using** box and select the communications port to be used to connect to the reference board.

The port selected is the same port that you connected to the serial cable from the reference board. Click `OK`.

Step 7. In the Port Settings tab, enter the following settings and click `OK`:

```
Bits per second = 19200
Data Bits = 8
Parity = None
Stop bits = 1
Flow control = XOn/XOff
```

Note

If the word *connected* does not appear in the lower left corner of the window, click `Call -> Connect` to establish the connection.

Step 8. Apply power to the board. The OS-9 bootstrap message is displayed.

Building the OS-9 ROM Image

Overview

The OS-9 ROM Image is a set of files and modules that collectively make up the OS-9 operating system. The specific ROM Image contents can vary from system to system depending on hardware capabilities and user requirements.

To simplify the process of loading and testing OS-9, the ROM Image is generally divided into two parts—the low-level image, called `coreboot`; and the high-level image, called `bootfile`.

Coreboot

The coreboot image is generally responsible for initializing hardware devices and locating the high-level (or bootfile) image as specified by its configuration. For example from a FLASH part, a harddisk, or Ethernet. It is also responsible for building basic structures based on the image it finds and passing control to the kernel to bring up the OS-9 system.

Bootfile

The bootfile image contains the kernel and other high-level modules (initialization module, file managers, drivers, descriptors, applications). The image is loaded into memory based on the device you select from the boot menu. The bootfile image normally brings up an OS-9 shell prompt, but can be configured to automatically start an application.

Microware provides a Configuration Wizard to create a coreboot image, a bootfile image, or an entire OS-9 ROM Image. The wizard can also be used to modify an existing image. The Configuration Wizard is automatically installed on your host PC during the Enhanced OS-9 installation process.

Using the Configuration Wizard

The OS-9 ROM Image enables booting from PCMCIA IDE type cards.

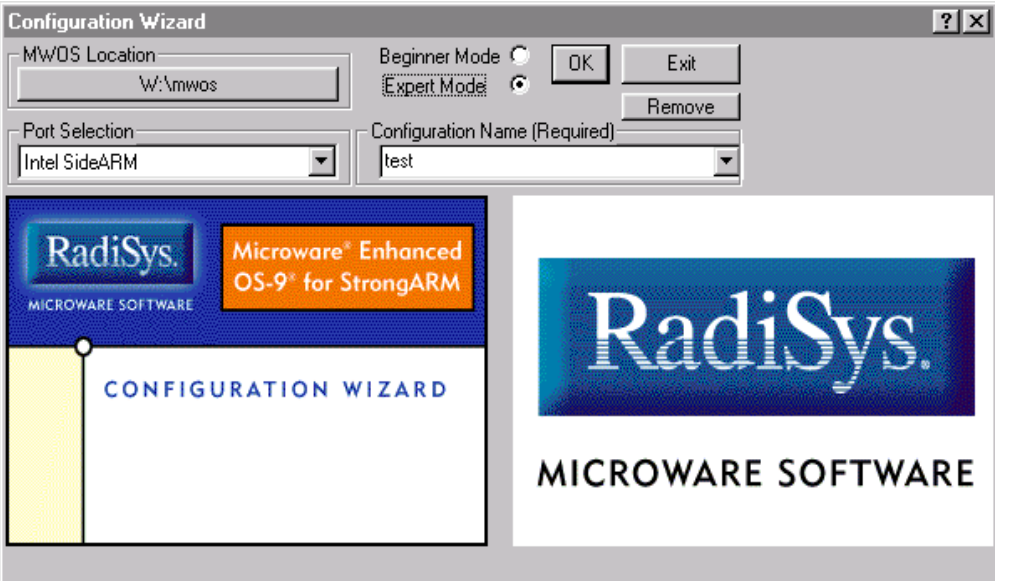
Note

Enhanced OS-9 for StrongARM supports ATA Flash cards.

To use the Configuration Wizard, perform the following steps:

Step 1. From the Windows desktop, select `Start -> Programs -> RadiSys -> Enhanced OS-9 for StrongARM -> Configuration Wizard`. You should see the following opening screen:

Figure 1-2 StrongARM Configuration Wizard

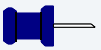


Step 2. Select the path where the MWOS directory structure is located from the **MWOS location** button.

Step 3. Select the target board from the **Port Selection** pull-down menu.

Step 4. Name the ROM image in the **Configuration Name** field.

Step 5. Select `Expert Mode` and click `OK`. The Main Configuration window is displayed.



Note

If you intend to use the Target board across a network, you need to configure the network settings.

Step 6. Select `Configure -> Build Image` to display the Master Builder window. If networking is desired, make sure the **SoftStax (SPF) Support** box is checked.

Step 7. Click `Build`. This builds a boot image that can be placed on the PCMCIA card.

Step 8. Insert the PCMCIA IDE card into the PCMCIA slot of your computer.

Step 9. Click `Save As` to save the file `os9kboot` to the root directory of the PCMCIA IDE card.

Step 10. Make sure the reference board is powered off and remove the PCMCIA IDE card from the computer.

WARNING

Inserting and removing a PCMCIA card with the power on can damage the card.

Step 11. Position the PCMCIA card so that the end with the connector holes is facing the PCMCIA socket and the label is facing up.

Step 12. Slide the card into the upper socket (socket 1) of the reference board until the card is securely seated into the connector.

Step 13. Apply power to the board. The reference board will boot from the IDE PCMCIA card and you should see the “$” prompt.

Creating a Startup File

When the Configuration Wizard is set to use a hard drive, or another fixed drive such as a PC Flash Card, as the default device, it automatically sets up the init module to call the `startup` file in the SYS directory in the target (For example: `/h0/SYS/startup`, `/mhc1/SYS/startup`). However, this directory and file will not exist until you create it. To create the startup file, complete the following steps:

Step 1. Create a SYS directory on the target machine where the `startup` file will reside (for example: `makdir /h0/SYS`, `makdir /dd/SYS`).

Step 2. On the host machine, navigate to the following directory:

`MWOS/OS9000/SRC/SYS`

In this directory, you will see several files. The files related to this section are listed below:

- `motd`: Message of the day file
- `password`: User/password file
- `termcap`: Terminal description file
- `startup`: Startup file

Step 3. Transfer all files to the newly created SYS directory on the target machine. (You can use Kermit, or FTP in ASCII mode to transfer these files.)

Step 4. Since the files are still in DOS format, you will be required to convert them into the OS-9 format with the `cudo` utility. The following command is an example:

```
cudo -cdo password
```

This will convert the `password` file from DOS to OS-9 format.

For More Information

For a complete description of all the `cudo` command options, refer to the ***Utilities Reference Manual*** located on the Enhanced OS-9 CD.

Step 5. Since the command lines in the startup file are system-dependent, it may be necessary to modify this file to fit your system configuration. It is recommended that you modify the file before transferring it to the target machine.

Example Startup File

Below is the example startup file as it appears in the `MWOS/OS9000/SRC/SYS` directory:

```
-tnxnp
tmode -w=1 nopause
*
*OS-9 - Version 3.0
*Copyright 2001 by Microware Systems Corporation
*The commands in this file are highly system dependent and
*should be modified by the user.
*
*setime </term              ;* start system clock
setime -s                   ;* start system clock
link mshell csl             ;* make "mshell" and "csl" stay in memory
* iniz r0 h0 d0 t1 p1 term  ;* initialize devices
* load utils                ;* make some utilities stay in memory
* tsmon /term /t1 &         ;* start other terminals
list sys/motd
setenv TERM vt100
tmode -w=1 pause
mshell<>>>/term -l&
```

For More Information

Refer to the **Making a Startup File** section in Chapter 9 of the ***Using OS-9*** manual for more information on startup files.

Optional Procedures

Connecting the Target to an Ethernet Network

Enhanced OS-9 for StrongARM supports using a 3COM Etherlink III - LAN PC Card for SoftStax TCP/IP connections. Also, Enhanced OS-9 for StrongARM provides system level support for telnet, FTP, and NFS.

To use Ethernet networking, you must create a bootfile that has the Ethernet options enabled and insert an Ethernet PCMCIA card into the reference board.

Step 6. From the Windows desktop, select `Start -> Programs -> RadiSys -> Enhanced OS-9 for StrongARM -> Configuration Wizard`. Click `OK` to open the Configuration Wizard with the name of your previously created configuration.

Step 7. Select `Configure -> Bootfile -> NetWork Configuration`. The Network Options dialog box appears.

Step 8. Change the network settings as needed. See the Configuration Wizard help for more information on the network settings.

Step 9. Select `Configure -> Build Image`. The **Master Builder** screen displays.

Step 10. On the **Master Builder** screen, select `SoftStax (SPF)` support and any other options you wish to use.

Step 11. Click `Build`. This builds a boot image that can be placed on the PCMCIA card.

Step 12. Insert the PCMCIA IDE card into the PCMCIA slot of your computer.

Step 13. Select `Save As` to save the file `os9kboot` to the root directory of the PCMCIA IDE card.

Step 14. Turn off the power to the reference board.

WARNING

Damage may occur to the PCMCIA card if it is inserted or removed while power is applied to the board.

Step 15. Position the PCMCIA IDE card so that the end with the PCMCIA female connector is facing PCMCIA socket 1 (the upper socket) and the label is facing up.

Step 16. Slide the PCMCIA IDE card into socket 1 (the upper socket) until the card is seated.

Step 17. Position the Ethernet PCMCIA card so that the end with the PCMCIA female connector is facing PCMCIA socket 0 (the lower socket) and the label is facing up.

Step 18. Slide the PCMCIA Ethernet card into socket 0 (the lower socket) until the card is seated.

Step 19. Restart your reference board.

Step 20. Test the Ethernet connection by pinging the reference board.

If the ping operation fails, you will have to check the following items:

- is the board connected to a live Ethernet port?
- is the Ethernet cable defective?
- are the network settings for the reference board correct?

Pinging the Reference Board

Windows 95, Windows 98, and Windows NT include a Ping command that can be used to test the Ethernet connection for the reference board.

Step 1. Go to the DOS prompt.

Step 2. Type `ping <IP Address>`.

The IP Address is the address you assigned to the evaluation board in either the Coreboot module or the Bootfile module. The address is typed without the <> brackets.

If the ping was successful, you will see the following response:

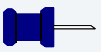
```
Reply from <IP Address>: bytes=xx time =xms TTL= xx
```

If the ping was unsuccessful, you will see the following response:

```
Request timed out.
```

Creating a new OS-9 Coreboot Image in Flash Memory

If you want to use ROM Ethernet services such as System State Debugging, you must create a new coreboot image. The coreboot image that was shipped with the reference board does not allow you to perform System State Debugging because the IP address in Flash ROM is set to “0.0.0.0”. You can create the coreboot image with an EPROM programmer.



Note

Re-creating the Coreboot image is required only when system state debugging is desired.

Making a Coreboot Image with an EPROM programmer

This section describes creating the Coreboot Image. When you are done creating the coreboot image, please refer to your EPROM programmer's instructions to learn how to load the Coreboot image into the EPROMS.

Step 1. Click the `Start` button on the Windows desktop.

Step 2. From the Windows desktop, select `Start -> Programs -> RadiSys -> Enhanced OS-9 for StrongARM -> Configuration Wizard`. The opening screen is displayed (see **Figure 1-2**).

Step 3. Give the boot image a name in the **Configuration Name** field.

Step 4. Select `Expert Mode` and click `OK`. The configuration screen is displayed.

Step 5. Make any necessary changes to the coreboot settings.

Step 6. Select `Configure -> Build Image` to display the Master Builder screen.

Step 7. Select the `Coreboot Only Image` setting and click `Build`.

Step 8. Click `Save As` to save the coreboot image to a directory of your choosing. If you do not have that directory on the drive, you can create it.

Step 9. Transfer the coreboot image to the EPROMS with the EPROM programmer. You will need to follow the documentation for the EPROM programmer to complete this step.

Chapter 2: Board-Specific Reference

This chapter contains information that is specific to the INTEL SA-1100 Microprocessor Reference Platform (SideARM) reference board. It includes the following sections:

- **Boot Options**
- **The Fastboot Enhancement**
- **OS-9 Vector Mappings**
- **SideARM GPIO Usage**

For More Information

For general information on porting OS-9, see the ***OS-9 Porting Guide.***

Boot Options

Following are the default boot options for the reference board. You can select these by hitting the space bar when the Now Trying to Override Autobooters message appears on the console port when booting.

You can configure these booters by altering the `default.des` file at the following location:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/ROM`

Booters can be configured to be either menu or auto booters. The auto booters automatically try and boot in order from each entry in the auto booter array. Menu booters from the defined menu booter array are chosen interactively from the console command line after getting the boot menu.

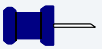
Booting from FLASH

When the `romcnfg.h` has a ROM search list defined the options `ro` and `lr` appear in the boot menu. If no search list is defined N/A appears in the boot menu. If an OS9 bootfile is programmed into flash in the address range defined in ports default.des file the system can boot and run from flash.

`ro`	rom boot—the system runs from the FLASH bank.
`lr`	load to ram—the system copies the flash image into ram and runs from there.

Booting from PCMCIA ATA Card

The system can boot from a PC formatted PCMCIA hard card which resides in slot 0 or slot 1.



Note

The system will hang during boot if there is not PCMCIA card, and it is configured to boot from one.

`ide1`	The file os9kboot is searched for in slot 1. If found it is copied to system RAM and runs from there.
`ide0`	The file os9kboot is searched for in slot 0. If found it is copied to system RAM and runs from there.

Booting from PCMCIA Ethernet Card

The system can boot using the BootP protocol using an Ethernet card and `eb` option.

`eb`	Ethernet boot—a PCMCIA card which supports ethernet will use the bootp protocol to transfer in a bootfile into RAM and the systems runs from there.

Booting over Serial Communications Port via kermit

The system can down-load a bootfile in binary form over its serial communication port at 115200 using the kermit protocol. The speed of this transfer depends of the size of the bootfile, but expect at least a 3 minute wait, dots will show the progress of the boot. The communications port is located at header J8 on and uses SP3.

`ker`	kermit boot—The os9kboot file is sent via the kermit protocol into system RAM and runs from there.

Down-load and Burn Coreboot or OS-9 bootfile to FLASH.

The system can down-load over the communication port the binary file coreboot or bootfile using the kermit protocol at 115200 baud, then burn this file into the Intel flash on the SideArm. This is a useful way to put your bootfile into flash, for executing from flash. The console port will output first dots as the file loads, then address as the file is burned into the flash. The communications port is located at header J8 on and uses SP3.

`dbc`	down-load and program the binary file coreboot into flash starting at address 0x08000000 up to 0x0803ffff (256k).
`dbb`	down-load and program the binary file os9kboot into flash starting at address 0x08040000 up to 0x08140000 (1 Meg).

Restart Booter

The restart booter allows a way to restart the bootstrap sequence.

`q`	quit—quit and attempt to restart the booting process.

Break Booter

The break booter allows entry to the system level debugger (if one exists). If the debugger is not in the system the system will reset.

`break`	break—break and enter the system level debugger rombug.

Example boot session and message.

```
OS-9 Bootstrap for the ARM

ATA IDE disk found in socket 00
Now trying to Override autobooters.

BOOTING PROCEDURES AVAILABLE ------------- <INPUT>

Boot embedded OS-9 in-place -------------- <N/A>
Copy embedded OS-9 to RAM and boot ------- <N/A>
Boot from PCMCIA-1 IDE ------------------- <ide1>
Boot from PCMCIA-0 IDE ------------------- <ide0>
Load bootfile via kermit Download -------- <ker>
Download and Program coreboot into FLASH - <dbc>
Download and Program bootfile into FLASH - <dbb>
Restart the System ----------------------- <q>
Enter system debugger -------------------- <break>

Select a boot method from the above menu: ide0


Wait for IDE drive ready.
IDE Model                  :       ATA_FLASH
Number Heads               : 0x0002
Total Cylinders            : 0x03d8
Sectors Per Track          : 0x0020

Checking Partitions        : 0
Fat Type                   : 0x16
File Name                  : OS9KBOOT
File Size                  : 0x000fdeb0
Start Cluster              : 0x00003a57
Reading Bootfile....

Boot Address               : 0xc002c850
Boot Size                  : 0x000fdeb0

OS-9 kernel was found.
A valid OS-9 bootfile was found.
$
```

The Fastboot Enhancement

The Fastboot enhancements to OS-9 provide faster system bootstrap performance to embedded systems. The normal bootstrap performance of OS-9 is attributable to its flexibility. OS-9 handles many different runtime configurations to which it dynamically adjusts during the bootstrap process.

The Fastboot concept consists of informing OS-9 that the defined configuration is static and valid. These assumptions eliminate the dynamic searching OS-9 normally performs during the bootstrap process and enables the system to perform a minimal amount of runtime configuration. As a result, a significant increase in bootstrap speed is achieved.

Overview

The Fastboot enhancement consists of a set of flags that control the bootstrap process. Each flag informs some portion of the bootstrap code that a particular assumption can be made and that the associated bootstrap functionality should be omitted.

The Fastboot enhancement enables control flags to be statically defined when the embedded system is initially configured as well as dynamically altered during the bootstrap process itself. For example, the bootstrap code could be configured to query dip switch settings, respond to device interrupts, or respond to the presence of specific resources which would indicate different bootstrap requirements.

In addition, the Fastboot enhancement's versatility allows for special considerations under certain circumstances. This versatility is useful in a system where all resources are known, static, and functional, but additional validation is required during bootstrap for a particular instance, such as a resource failure. The low-level bootstrap code may respond to some form of user input that would inform it that additional checking and system verification is desired.

Implementation Overview

The Fastboot configuration flags have been implemented as a set of bit fields. An entire 32-bit field has been dedicated for bootstrap configuration. This four-byte field is contained within the set of data structures shared by the ModRom sub-components and the kernel. Hence, the field is available for modification and inspection by the entire set of system modules (high-level and low-level). Currently, there are six bit flags defined with eight bits reserved for user-definable bootstrap functionality. The reserved user-definable bits are the high-order eight bits (31-24). This leaves bits available for future enhancements. The currently defined bits and their associated bootstrap functionality are listed below:

B_QUICKVAL

The `B_QUICKVAL` bit indicates that only the module headers of modules in ROM are to be validated during the memory module search phase. This causes the CRC check on modules to be omitted. This option is a potential time saver, due to the complexity and expense of CRC generation. If a system has many modules in ROM, where access time is typically longer than RAM, omitting the CRC check on the modules will drastically decrease the bootstrap time. It is rare that corruption of data will ever occur in ROM. Therefore, omitting CRC checking is usually a safe option.

B_OKRAM

The `B_OKRAM` bit informs both the low-level and high-level systems that they should accept their respective RAM definitions without verification. Normally, the system probes memory during bootstrap based on the defined RAM parameters. This allows system designers to specify a possible RAM range, which the system validates upon startup. Thus, the system can accommodate varying amounts of RAM. In an embedded system where the RAM limits are usually statically defined and presumed to be functional, however, there is no need to validate the defined RAM list. Bootstrap time is saved by assuming that the RAM definition is accurate.

B_OKROM

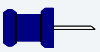
The B_OKROM bit causes acceptance of the ROM definition without probing for ROM. This configuration option behaves like the B_OKRAM option, except that it applies to the acceptance of the ROM definition.

B_1STINIT

The B_1STINIT bit causes acceptance of the first init module found during cold-start. By default, the kernel searches the entire ROM list passed up by the ModRom for init modules before it accepts and uses the init module with the highest revision number. In a statically defined system, time is saved by using this option to omit the extended init module search.

B_NOIRQMASK

The B_NOIRQMASK bit informs the entire bootstrap system that it should not mask interrupts for the duration of the bootstrap process. Normally, the ModRom code and the kernel cold-start mask interrupts for the duration of the system startup. However, some systems that have a well defined interrupt system (i.e. completely calmed by the sysinit hardware initialization code) and also have a requirement to respond to an installed interrupt handler during system startup can enable this option to prevent the ModRom and the kernel cold-start from disabling interrupts. This is particularly useful in power-sensitive systems that need to respond to “power-failure” oriented interrupts.



Note

Some portions of the system may still mask interrupts for short periods during the execution of critical sections.

B_NOPARITY

If the RAM probing operation has not been omitted, the B_NOPARITY bit causes the system to not perform parity initialization of the RAM. Parity initialization occurs during the RAM probe phase. The B_NOPARITY option is useful for systems that either require no parity initialization at all or systems that only require it for “power-on” reset conditions. Systems that only require parity initialization for initial “power-on” reset conditions can dynamically use this option to prevent parity initialization for subsequent “non-power-on” reset conditions.

Implementation Details

This section describes the compile-time and runtime methods by which the bootstrap speed of the system can be controlled.

Compile-time Configuration

The compile-time configuration of the bootstrap is provided by a pre-defined macro (BOOT_CONFIG), which is used to set the initial bit-field values of the bootstrap flags. You can redefine the macro for recompilation to create a new bootstrap configuration. The new over-riding value of the macro should be established by redefining the macro in the rom_config.h header file or as a macro definition parameter in the compilation command.

The rom_config.h header file is one of the main files used to configure the ModRom system. It contains many of the specific configuration details of the low-level system. Below is an example of how you can redefine the bootstrap configuration of the system using the BOOT_CONFIG macro in the rom_config.h header file:

```
#define BOOT_CONFIG (B_OKRAM + B_OKROM + B_QUICKVAL)
```

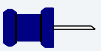
Below is an alternate example showing the default definition as a compile switch in the compilation command in the makefile:

```
SPEC_COPTS = -dNEWINFO –dNOPARITYINIT –dBOOT_CONFIG=0x7
```

This redefinition of the `BOOT_CONFIG` macro results in a bootstrap method that accepts the RAM and ROM definitions without verification, and also validates modules solely on the correctness of their module headers.

Runtime Configuration

The default bootstrap configuration can be overridden at runtime by changing the `rinf->os->boot_config` variable from either a low-level P2 module or from the `sysinit2()` function of the `sysinit.c` file. The runtime code can query jumper or other hardware settings to determine what user-defined bootstrap procedure should be used. An example P2 module is shown below.



Note

If the override is performed in the `sysinit2()` function, the effect is not realized until after the low-level system memory searches have been performed. This means that any runtime override of the default settings pertaining to the memory search must be done from the code in the P2 module code.

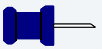
```
#define NEWINFO
#include <rom.h>
#include <types.h>
#include <const.h>
#include <errno.h>
#include <romerrno.h>
#include <p2lib.h>

error_code p2start(Rominfo rinf, u_char *glbls)
{
    /* if switch or jumper setting is set… */
    if (switch_or_jumper == SET) {
        /* force checking of ROM and RAM lists */
        rinf->os->boot_config &= ~(B_OKROM+B_OKRAM);
    }
    return SUCCESS;
}
```

OS-9 Vector Mappings

This section contains the vector mappings for the OS-9 SideARM/SideKick implementation of the SA1100.

The ARM standard defines exceptions 0x0-0x8. The OS-9 system maps these 1-1. External interrupts from vector 0x6 are expanded to the virtual vector rage shown below by the `irq1100` module and `irq1101` modules.



Note

Vectors can be virtually remapped from a ROM at physical address 0, into DRAM at virtual address 0. This speeds up interrupt response time and is enabled by defining the first cache list entry as a sub 1 Meg size.

For More Information

See the 1100/1101 hardware documentation for more information on individual sources.

Table 2-1 and **Table 2-2** show the OS-9 IRQ assignment for the StrongARM SA1100 board.

Table 2-1 IRQ Assignments and ARM Functions

OS-9 IRQ #	ARM Function
0x0	Processor Reset
0x1	Undefined Instruction

Table 2-1 IRQ Assignments and ARM Functions (continued)

OS-9 IRQ #	ARM Function
0x2	Software Interrupt
0x3	Abort on Instruction Prefetch
0x4	Abort on Data Access
0x5	Unassigned/Reserved
0x6	External Interrupt
0x7	Fast Interrupt
0x8	Alignment error

Table 2-2 IRQ Assignments and SA1100 Specific Functions

OS-9 IRQ #	SA1100 Specific Function (pic)
0x40	GPIO[0] Edge Detect
0x41	GPIO[1] Edge Detect
0x42	GPIO[2] Edge Detect
0x43	GPIO[3] Edge Detect
0x44	GPIO[4] Edge Detect
0x45	GPIO[5] Edge Detect
0x46	GPIO[6] Edge Detect

Table 2-2 IRQ Assignments and SA1100 Specific Functions (continued)

OS-9 IRQ #	SA1100 Specific Function (pic)
0x47	GPIO[7] Edge Detect
0x48	GPIO[8] Edge Detect
0x49	GPIO[9] Edge Detect
0x4a	GPIO[10] Edge Detect
0x4b	OR of GPIO edge detects 27 - 11
0x4c	LCD controller service request
0x4d	UDC service request (0)
0x4e	SDLC service request (1a)
0x4f	UART service request (1b)
0x50	UART/HSSP service request (2)
0x51	UART service request (3)
0x52	MCP service request (4a)
0x53	SSP service request (4b)
0x54	DMA controller channel 0
0x55	DMA controller channel 1
0x56	DMA controller channel 2
0x57	DMA controller channel 3

Table 2-2 IRQ Assignments and SA1100 Specific Functions (continued)

OS-9 IRQ #	SA1100 Specific Function (pic)	
0x58	DMA controller channel 4	
0x59	DMA controller channel 5	
0x5a	OS timer 0	
0x5b	OS timer 1	
0x5c	OS timer 2	
0x5d	OS timer 3	
0x5e	One Hz clock tick	
0x5f	RTC als alarm register	
0x60	GPIO[11] Edge Detect	*The vector 0x4b OR is broken out here to make each one distinct
0x61	GPIO[12] Edge Detect	
0x62	GPIO[13] Edge Detect	
0x63	GPIO[14] Edge Detect	
0x64	GPIO[15] Edge Detect	
0x65	GPIO[16] Edge Detect	
0x66	GPIO[17] Edge Detect	
0x67	GPIO[18] Edge Detect	

Table 2-2 IRQ Assignments and SA1100 Specific Functions (continued)

OS-9 IRQ #	SA1100 Specific Function (pic)	
0x68	GPIO[19] Edge Detect	
0x69	GPIO[20] Edge Detect	
0x6a	GPIO[21] Edge Detect	
0x6b	GPIO[22] Edge Detect	
0x6c	GPIO[23] Edge Detect	
0x6d	GPIO[24] Edge Detect	*Input for SideKick 1101 IRQ
0x6e	GPIO[25] Edge Detect	
0x6f	GPIO[26] Edge Detect	
0x70	GPIO[27] Edge Detect	

Table 2-3 IRQ Assignments and SA1101 Specific Functions

OS-9 IRQ #	SA1101 Specific Function
0x71	GPIOA[0] Edge Detect
0x72	GPI0A[1] Edge Detect
0x73	GPIOA[2] Edge Detect
0x74	GPIOA[3] Edge Detect

Table 2-3 IRQ Assignments and SA1101 Specific Functions

OS-9 IRQ #	SA1101 Specific Function
0x75	GPIOA[4] Edge Detect
0x76	GPIOA[5] Edge Detect
0x77	GPIOA[6] Edge Detect
0x78	GPIOA[7] Edge Detect
0x79	GPIOB[0] Edge Detect
0x7a	GPIOB[1] Edge Detect
0x7b	GPIOB[2] Edge Detect
0x7c	GPIOB[3] Edge Detect
0x7d	GPIOB[4] Edge Detect
0x7e	GPIOB[5] Edge Detect
0x7f	GPIOB[6] Edge Detect
0x80	Reserved
0x81	KPXIn[0] Keypad X
0x82	KPXIn[1] Keypad X
0x83	KPXIn[2] Keypad X
0x84	KPXIn[3] Keypad X
0x85	KPXIn[4] Keypad X

Table 2-3 IRQ Assignments and SA1101 Specific Functions

OS-9 IRQ #	SA1101 Specific Function
0x86	KPXIn[5] Keypad X
0x87	KPXIn[6] Keypad X
0x88	KPXIn[7] Keypad X
0x89	KPYIn[0] Keypad Y
0x8a	KPYIn[1] Keypad Y
0x8b	KPYIn[2] Keypad Y
0x8c	KPYIn[3] Keypad Y
0x8d	KPYIn[4] Keypad Y
0x8e	KPYIn[5] Keypad Y
0x8f	KPYIn[6] Keypad Y
0x90	KPYIn[7] Keypad Y
0x91	KPYIn[8] Keypad Y
0x92	KPYIn[9] Keypad Y
0x93	KPYIn[10] Keypad Y
0x94	KPYIn[11] Keypad Y
0x95	KPYIn[12] Keypad Y
0x96	KPYIn[13] Keypad Y

Table 2-3 IRQ Assignments and SA1101 Specific Functions

OS-9 IRQ #	SA1101 Specific Function
0x97	KPYIn[14] Keypad Y
0x98	KPYIn[15] Keypad Y
0x99	MsTxInt PS2 Mouse
0x9a	MsRxInt PS2 Mouse
0x9b	TpTxInt PS2 Trackpad
0x9c	TpRxInt PS2 Trackpad
0x9d	IntReqTrc IEEE 1284
0x9e	IntReqTim IEEE 1284
0x9f	IntReqRav IEEE 1284
0xa0	IntReqInt IEEE 1284
0xa1	IntReqEmp IEEE 1284
0xa2	IntReqDat IEEE 1284
0xa3	VideoInt VGA controller
0xa4	FifoInt Update fifo
0xa5	nIrqHciM USB
0xa6	IrqHciBuffAcc USB
0xa7	IrqHciRmtWkp USB

Table 2-3 IRQ Assignments and SA1101 Specific Functions

OS-9 IRQ #	SA1101 Specific Function
0xa8	nHciMFCIr USB
0xa9	USBError USB
0xaa	S0_Ready_nIREQ PCMCIA
0xab	S1_Ready_nIREQ PCMCIA
0xac	S0_CDValid PCMCIA
0xad	S1_CDValid PCMCIA
0xae	S0_BVD1_STSCHG PCMCIA
0xaf	S1_BVD1_STSCHG PCMCIA
0xb0	USB Wakeup USB

Note

Fast Interrupt Vector (0x7)

The ARM4 defined fast interrupt (FIQ) mapped to vector 0x7 is handled differently by the OS-9 interrupt code and can not be used as freely as the external interrupt mapped to vector 0x6. To make fast interrupts as quick as possible for extremely time critical code, no context information is saved on exception and FIQs are never masked. This requires any exception handler to save and restore its necessary context if the FIQ mechanism is to be used. This requirement means that a FIQ handler's entry and exit points must be in assembly, as the C compiler will make assumptions about context. In addition, no system calls are possible unless a full C ABI context save has been done first. The OS-9 IRQ code for the SA1100 has assigned all interrupts as normal external interrupts and the user must re-define a source as an FIQ to make use of this feature.

SideARM GPIO Usage

Table 2-4 shows GPIO usage of the SideARM board in an OS-9 system.

For More Information

See the Intel SideARM board guide for available alternate pin functions.

Table 2-4 GPIO Usage of SideARM Board

GPIO	Signal Name	Direct	Description
GPIO0	SW2	Input	SideARM switch SW22
GPIO1	SW1	Input	SideARM switch SW21
GPIO2	P0_STSCHG	Input	PCMCIA Slot 0 status change
GPIO3	P0_IRQ	Input	PCMCIA Slot 0 IRQ
GPIO4	P0_F1	Input	PCMCIA Slot 0 valid
GPIO5	P1_STSCHG	Input	PCMCIA Slot 1 status change
GPIO6	P1_IRQ	Input	PCMCIA Slot 1 IRQ
GPIO7	P1_F	Input	PCMCIA Slot 1 valid
GPIO8	LED_GRN2	Output	

Table 2-4 GPIO Usage of SideARM Board (continued)

GPIO	Signal Name	Direct	Description
GPIO9	LED_GRN1	Output	
GPIO10	SSP_TXD	Output	SSP Port transmit
GPIO11	SSP_RXD	Input	SSP Port Receive
GPIO12	SSP_SCLK	Output	SSP Port Clock
GPIO13	SSP_SFRM	Output	SSP Port Frame
GPIO14	UART_TXD	Output	SP1 uart transmit
GPIO15	UART_RXD	Input	SP1 uart receive
GPIO16	SDLC_HSKO	Output	
GPIO17	SDLC_AAF	Output	
GPIO18	SDLC_HSKI	Input	
GPIO19	SDLC_GPI	Input	
GPIO20	LED_RED	Output	led output
GPIO21	IRDA_SD	Output	IRDA data line
GPIO22	IRQ_C	Input	
GPIO23	KBC_WKUP	Output	Keyboard wake up
GPIO24	KBC_WUKO	In/Out	Keyboard wake up
GPIO25	KBC_ATN	Input	Keyboard atn

Table 2-4 GPIO Usage of SideARM Board (continued)

GPIO	Signal Name	Direct	Description
GPIO26	RCLK_OUT	Output	Ref clock output
GPIO27	32Khz Out	Output	32Khz Out clock

GPIO Interrupt Polarity

When GPIOs are used as interrupt sources, the `_PIC_ENABLE()` function will set default polarity to rising edge (GRER) along with enabling the interrupt at the SA1100 PIC. If falling edge is required, software must assert the appropriate bit in the GFER and negate the corresponding bit in the GRER. The polarity of the SA1101's interrupt/gpio sources is also set by the `_PIC_ENABLE()` function call. They are defined to the OS-9 defaults of `INTPOL0=0x00000000` and `INTPOL1=0x7E801000`. If other values are needed software must set the corresponding bits.

Port Specific Utilities

The following port specific utility is included:

- `ucbtouch`

ucbtouch

Syntax

```
ucbtouch <>
```

Description

The `ucbtouch` utility prints the raw x,y and pressure values at a set sample rate.

Press the touch screen and observe the output on your console. The utility is helpful in determining whether your touch screen is connected properly.

Example

```
$ ucbtouch
Touch[00000]: Touch=0x30c3 X1=00328 Y1=00321 P= 28 X=329 Y=322
Touch[00001]: Touch=0x30c3 X1=00329 Y1=00325 P= 28 X=330 Y=326
Touch[00002]: Touch=0x30c3 X1=00329 Y1=00321 P= 28 X=330 Y=322
Touch[00003]: Touch=0x30c3 X1=00329 Y1=00321 P= 29 X=330 Y=322
Touch[00004]: Touch=0x30c3 X1=00329 Y1=00319 P= 29 X=330 Y=320
Touch[00005]: Touch=0x30c3 X1=00329 Y1=00321 P= 28 X=330 Y=322
Touch[00006]: Touch=0x30c3 X1=00329 Y1=00327 P= 28 X=330 Y=328
Touch[00007]: Touch=0x30c3 X1=00329 Y1=00321 P= 28 X=330 Y=322
Touch[00008]: Touch=0x30c3 X1=00329 Y1=00321 P= 29 X=330 Y=322
Touch[00009]: Touch=0x30c3 X1=00329 Y1=00322 P= 28 X=330 Y=323
Touch[00010]: Touch=0x30c3 X1=00329 Y1=00319 P= 28 X=0 Y=0
Touch[00011]: Touch=0x30c3 X1=00328 Y1=00321 P= 28 X=-1 Y=2
Touch[00012]: Touch=0x30c3 X1=00329 Y1=00315 P= 28 X=0 Y=-4
Touch[00013]: Touch=0x30c3 X1=00329 Y1=00322 P= 29 X=0 Y=3
```

Appendix A: Board-Specific Modules

This chapter describes the modules specifically written for the target board. It includes the following sections:

- **Low-Level System Modules**
- **High-Level System Modules**

Low-Level System Modules

For More Information

For a complete list of OS-9 modules common to all boards, see the ***OS-9 Device Descriptor and Configuration Module Reference*** manual.

The following low-level system modules are tailored specifically for the Intel SA1100 SideARM platform. The functionality of these modules can be altered through changes to the configuration data module (`cnfgdata`). **Table A-1** provides a list and brief description of the modules.

These modules can be found in the following directory:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/ROM`

Table A-1 SideARM-Specific Low-Level System Modules

Module Name	Description
`cnfgdata`	Contains the low-level configuration data.
`cnfgfunc`	Provides access services to `cnfgdata` data.
`commcnfg`	Inits communication port defined in `cnfgdata`.
`conscnfg`	Inits console port defined in `cnfgdata`.
`ide`	IDE boot support module. PCMCIA compatible.
`io1100`	Provides polled serial driver support for the low-level system.

Table A-1 SideARM-Specific Low-Level System Modules (continued)

Module Name	Description
`llcis`	Inits the PCMCIA interface including cards.
`lle509`	Provides low-level ethernet services via 3COM PCMCIA card.
`portmenu`	Inits booters defined in the `cnfgdata`.
`romcore`	Board specific initialization code.
`tmr1_1100`	Provides low-level timer services via time base register.
`usedebug`	Inits low-level debug interface to RomBug, SNDP, or none.
`skinit`	Inits the SA1101 and its interface to the SA1100.

The following low-level system modules provide generic services for OS9000 Modular ROM. **Table A-2** provides a list and brief description of the modules.

These modules can be found in the following directory:

`MWOS/OS9000/ARMV3/CMDS/BOOTOBJS/ROM`

Table A-2 Generic Services Low-Level System Modules

Module Name	Description
`bootsys`	Booter registration service module.
`console`	Provides console services.
`dbgentry`	Inits debugger entry point for system use.

Table A-2 Generic Services Low-Level System Modules (continued)

Module Name	Description
dbgserve	Provides debugger services.
excption	Provides low-level exception services.
flshcach	Provides low-level cache management services.
flashb	Booter which provides low-level flash programming.
28f016	Provides 28f016 specific hooks for use with flashb.
hlproto	Provides user level code access to protoman.
llbootp	Booter which provides bootp services.
llip	Provides low-level IP services.
llslip	Provides low-level SLIP services.
lltcp	Provides low-level TCP services.
lludp	Provides low-level UDP services.
llkermit	Booter which uses kermit protocol.
notify	Provides state change information for use with LL and HL drivers.
override	Booter which allows choice between menu and auto booters.
parser	Provides argument parsing services.
pcman	Booter which reads MS-DOS file system.

Table A-2 Generic Services Low-Level System Modules (continued)

Module Name	Description
`protoman`	Protocol management module.
`restart`	Booter which causes a soft reboot of system.
`romboot`	Booter which allows booting from ROM.
`rombreak`	Booter which calls the installed debugger.
`rombug`	Low-level system debugger.
`sndp`	Provides low-level system debug protocol.
`srecord`	Booter which accepts S-Records.
`swtimer`	Provides timer services via software loops.

High-Level System Modules

The following OS-9 system modules are tailored specifically for your Intel SA1100 SideArm/SideKick board peripherals. Unless otherwise specified, each module can be found in a file of the same name in the following directory:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS`

CPU Support Modules

These files are located in the following directory:

`MWOS/OS9000/ARMV4/CMDS/BOOTOBJS`

`kernel`	The kernel provides all basic services for the OS-9 system.
`cache`	Provides cache control for the CPU cache hardware. The cache module is in the file `cach1100`.
`fpu`	Provides software emulation for floating point instructions.
`ssm`	The System Security Module provides support for the Memory Management Unit (MMU) on the CPU.
`vectors`	Provides interrupt service entry and exit code. The vectors module is found in the file `vect110`.

System Configuration Module

These files are located in the following directory:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/INITS`

`init`	Descriptor module with high-level system initialization information.
`nodisk`	Same as init, but used in a disk-less system.

Power Management Support Modules

These modules provide an interface to control the power states of SideARM devices.

The supported SA1100 CPU power states are SLEEP, IDLE, RUN.

`pwrman`	`P2module` that provides generic power management functions.
`pwrplcy`	`P2module` that provides power state control functions.
`sysif`	`P2module` that provides SA1100 CPU power state control.

Interrupt Controller Support

This module provides extensions to the vectors module by mapping the single interrupt generated by an interrupt controller into a range of pseudo vectors which are recognized by OS-9 as extensions to the base CPU exception vectors.

For More Information

The mappings are described in **Chapter 2**.

`irq1100`	`P2module` that provides interrupt acknowledge and dispatching support for the SA1100 pic (vector range 0x40 - 0x70).
`IRQ1101`	`P2module` which provides interrupt acknowledge and dispatching support for the SA1101 pic (vector range 0x71 - 0xb0).

Real Time Clock

`rtc1100`	Driver that provides OS-9 access to the SA1100 on-board real time clock.

Ticker

`tk1100`	Driver that provides the system ticker based on the SA1100 Operating System Timer.

Abort Handler

`abort`	`P2module` which provides a way to enter the system-state debugger via the GPIO[0] interrupt triggered by SideARM switch S1[8].

Generic IO Support modules (File Managers)

These files are located in the following directory:

`MWOS/OS9000/ARMV3/CMDS/BOOTOBJS`

`ioman`	Provides generic io support for all IO device types.
`scf`	Provides generic character device management functions.

`rbf`	Provides generic block device management functions for OS-9 specific format.
`pcf`	Provides generic block device management functions for MS-DOS FAT format.
`spf`	Provides generic protocol device management function support.
`mfm`	Provides generic graphics device support for MAUI.
`pipeman`	Provides a memory FIFO buffer for communication.

Pipe Descriptor

This file is located in the following directory:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/DESC`

`pipe`	Pipeman descriptor that provides a RAM based FIFO which can be used for process communication.

RAM Disk Support

`ram`	RBF driver which provides a RAM based virtual block device.

Descriptors for Use with RAM

These files are located in the following directory:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/DESC/RAM`

`r0`	RBF descriptor which provides access to a ram disk.
`r0.dd`	Same as `r0` except with module name dd (for use as the default device).

Serial and Console Devices

`sc1100`	SCF driver which provides serial support the SA1100's SP1 and SP3 ports when configured as UARTS.

Descriptors for Use with sc1100

`term1/t1`	Descriptor modules for use with sc1100 and SP1. SideArm Board header:J8 Default Baud Rate:19200 Default Parity:None Default Data Bits:8 Default Handshake:Software
`term3/t3`	Descriptor modules for use with sc1100 and SP3. SideArm Board header:J11 Default Baud Rate:115200 Default Parity:None Default Handshake:Software
`m0`	Serial mouse descriptor module for use with sc1100 and SP1. SideARM Board header:J8 Default Baud Rate:1200 Default Parity:None Default Data Bits:8 Default Handshake:None

WARNING
Power must be wired to PIN 7 for proper operation.

`sc1101` SCF driver which provides raw ps/2 mouse and keyboard support via the SA1101's PS/2 ports.

Descriptors for Use with sc1101

`m0/m1` Descriptor modules for use with sc1101 and the SA1101's PS/2 ports.

SideKick Board header:J8/J6

Scan codes:System Scan codes when using keyboard.

Standard PS/2 mouse.

`scllio` SCF driver which provides serial support via the polled low-level serial driver.

Descriptors for Use with scllio

`vcons/term` Descriptor modules for use with scllio in conjunction with a low-level serial driver. Port configuration and set up follows what is configured in cnfgdata for the console port. It is possible for `scllio` to communicate with a true low-level serial device driver like io1100, or with an emulated serial interface provided by `iovcons`. See the OEM manual for more information.

PCMCIA Support for IDE Type Devices

`rb1003`	RBF/PCF driver which provides driver support for IDE/EIDE devices. This driver is used to provide disk support for PCMCIA ATA FLASH.

Descriptors for use with rb1003

`hc1/hc1fmt and hc1.dd`

RBF Descriptor modules for use with PCMCIA slot #0 (top)

SideKick Board header:J1

`hc1fmt:` format enabled

`hc1.dd:` module name of dd

`mhc1/mhc1.dd` PCF Descriptor modules for use with PCMCIA

SideKick Board header:J1

`mhc1.dd:` module name of dd

`he1/he1fmt and he1.dd`

RBF Descriptor modules for use with PCMCIA slot #1 (bottom)

SideKick Board header: J1

`he1fmt:` format enabled

`he1.dd:` module name of dd

`mhe1/mhe1.dd` PCF Descriptor modules for use with PCMCIA slot #1 (bottom)

SideKick Board header:J1

mhc1.dd: module name of dd

PCMCIA Support for 3COM Ethernet Card

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/SPF`

`spe509_pcm`	SPF driver to support ethernet for a 3COM EtherLink III PCMCIA card.

Descriptors for Use with rb1003

`spe30`	SPF descriptor modules for use with PCMCIA slot #0 (top, J1)
`spe31`	SPF descriptor modules for use with PCMCIA slot #1 (bottom, J1)

Network Configuration Modules

`inetdb/inetdb2/rpcdb`

UCB1200 Support Modules

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/SPF`

`spucb1200`	SPF driver which support the on-board Phillips UCB1200 chip. This device communicates to the SA1100 over SP4 using MCP.

Descriptors for Use with spucb1200

`ucb`	SPF descriptor module which provides access to UCB1200.
`ucb_touch`	SPF descriptor module which is used with a screen.

Maui Graphical Support Modules (VGA)

`MWOS/OS9000/ARMV4/PORTS/SIDEARM/CMDS/BOOTOBJS/MAUI`

`gx_sa1101`	MFM MAUI driver module with support for the SideKick VGA.

Descriptors for Use with gx_sa1101

`vga`	MFM MAUI descriptor module for SideKick VGA. Resolutions: 640x480,800x600,1024x768
`sd_ucb1200`	MFM MAUI driver module which provides PCM/mu-law sound support via the ucb1200. A speaker and microphone are not provided on the standard SideARM board. These parts need to be added to make use of sound functions.

Descriptors for Use with sd_ucb1200

`snd`	MFM MAUI descriptor module for UCB1200 sound functions.

MAUI Configuration Modules

`cdb_sa`	Maui configuration data base module for a SideARM board implementation.
`cdb_sk`	Maui configuration data base module for a SideKick VGA.
`cdb_ps2`	PS/2 mouse configuration data base module.
`cdb_smouse`	Serial mouse configuration data base module.
`cdb_touch`	Touch screen configuration data base module (for LCD).

For More Information

The MAUI drivers are described in more detail in **Appendix B: MAUI Driver Descriptions**.

Appendix B: MAUI Driver Descriptions

This chapter provides MAUI driver descriptions. It includes the following sections:

- **SideARM Objects**
- **GX_SA1101 VGA Graphic Driver Specification**
- **SD_UCB1200 Sound Driver Specification**
- **SPUCB1200 driver for the UCB1200 Codec**
- **MP_UCB1200 MAUI Touch Screen Protocol Module**

SideARM Objects

This package provides object-level support for the Intel Brutus reference board. The port directory is at the following location:

`MWOS/OS9000/ARMV4/PORTS/SIDEARM`

MAUI objects

`cdb`	Lists the devices on the system; parts are listed so different device sets can be specified in the configuration wizard
`cdb_sa`	required base CDB listing SideArm devices
`cdb_sk`	optional CDB listing additional SideKick devices
`cdb_smouse`	optional CDB if using a serial mouse
`cdb_ps2`	optional CDB if using PS2 devices on SideKick
`mp_bsptr`	Bus mouse serial protocol module
`mp_msptr`	Serial mouse protocol module.
`mp_ucb1200`	Touch screen protocol module for the UCB1200.
`mp_xtkbd`	XT scan code keyboard protocol module
`vga and gx_sa1101`	VGA graphics descriptor and driver.

GX_SA1101 VGA Graphic Driver Specification

This section describes the hardware specification of the StrongARM/SideKick SA1101 VGA driver (`gx_sa1101`) and descriptor (`vga`). The hardware sub-type defines the board configuration. This specification should be used with the MAUI Graphics Device API.

Device Capabilities

Information about the hardware capabilities is determined by calling `gfx_get_dev_cap()`. The hardware sub-type defines the board configuration. This function returns a data structure formatted as shown in **Table B-1**. See `GFX_DEV_CAP` for more information about this data structure.

Table B-1 gfx_get_dev_cap() Data Structure

Member Name	Description	Value
`hw_type`	Hardware type (embedded in driver)	SA1101 VGA Controller
`hw_subtype`	Hardware subtype (embedded in descriptor)	Sidekick VGA Controller w/ IOBLT
`sup_vpmix`	Supports viewport mixing	FALSE
`sup_extvid`	Supports external video as a backup	FALSE
`sup_bkcol`	Supports background color	TRUE
`sup_vptrans`	Supports viewport transparency	FALSE

Table B-1 gfx_get_dev_cap() Data Structure (continued)

Member Name	Description	Value
sup_vpinten	Supports viewport intensity	FALSE
sup_sync	Supports retrace synchronization	TRUE
num_res	Number of display resolutions	3
res_info	Array of display resolution information	See Display Resolution table
dac_depth	Depth of the DAC in bits	12
num_cm	Number of coding methods	1
cm_info	Array of coding method information	See Coding Methods table
sup_viddecode	Supports video decoding into a drawmap	FALSE

Display Resolution

The following display resolutions are supported by this driver. The first row is the default and is set when gfx_open_dev() is called. The resolution may be changed by calling gfx_set_display_size().

Table B-2 Display Specifications

Width	Height	Refresh Rate	Interlace Mode	Aspect Ratio X:Y
640	480	72.8	GFX_INTL_OFF	1:1
800*	600	72.8	GFX_INTL_OFF	1:1
1024*	768	70.4	GFX_INTL_OFF	1:1

*Dedicated memory mode only

Coding Methods

The Sidekick supports only 8-bit CLUT.

Table B-3 Coding Method Description

Coding Method	CLUT Based	X,Y Multipliers	Palette Color Types
GFX_CM_8BIT	TRUE	1,1	GFX_COLOR_RGB

Viewport Complexity

The driver supports one active viewport at a time. The application can create multiple viewports and stack them. The viewport must be aligned with, and the same size as the display. Display drawmaps must be the same size as the viewport.

Memory

Applications are expected to request graphics memory from the driver. The driver allocates memory from the system as needed. It requests this memory from color 0x80. This memory (specified in the init module) is located at the bottom of 16 MB DRAM address space and is marked as non cached.

The driver can operate in dedicated mode.

Location

This driver's source is located in:

`SRC/DPIO/MFM/DRVR/GX_SA1101`

This driver's makefiles are located in:

`OS9000/ARMV4/PORTS/SIDEARM/MAUI/GX_SA1101`

This directory contains the makefiles and descriptor header file to build the descriptor(s) and driver(s) (not all packages include driver source) for the StrongARM reference platform. This directory contains:

`makefile`	Calls each of the other makefiles in this directory
`drvr.mak`	Builds the driver
`desc.mak`	Builds the descriptor(s)
`mfm_desc.h`	Defines values for all modifiable fields of the descriptor(s)

Build the Driver

The driver source is located in SRC/DPIO/MFM/DRVR/GX_SA1101. To build the driver, use the following commands:

```
cd OS9000/ARMV4/PORTS/SIDEARM/MAUI/GX_SA1101
os9make -f drvr.mak
```

Build the Descriptor

To build a new descriptor, modify mfm_desc.h, and use the following commands to compile:

```
cd OS9000/ARMV4/PORTS/SIDEARM/MAUI/GX_SA1101
os9make -f desc.mak
```

To build both the driver and the descriptor you can specify os9make with no parameters.

SD_UCB1200 Sound Driver Specification

This section describes the hardware specifications for the Philips UCB1200 driver `sd_ucb1200`. The hardware sub-type defines the board configuration. This specification should be used in conjunction with the MAUI Sound Driver Interface.

This driver works in conjunction with the spucb1200 driver.

Device Capabilities

Information about the hardware capabilities is determined by calling `_os_gs_snd_devcap()`. This function returns a data structure formatted as in the following table. See `SND_DEV_CAP` for more information about this data structure.

Table B-4 Data Returned in SND_DEV_CAP

Member Name	Value	Description
`hw_type`	"CS4231	"Hardware type
`hw_subtype`	"CS4231A	"Hardware sub-type
`sup_triggers`	`SND_TRIG_ANY`	Supported triggers
`play_lines`	`SND_LINE_SPEAKER`	Play gain/mix lines
`record_lines`	`SND_LINE_MIC`	Record gain/mix lines
`sup_gain_cmds`	`SND_GAIN_CMD_MONO`	Mask of supported gain commands
`num_gain_caps`	2	Number of `SND_GAIN_CAPs`

Table B-4 Data Returned in SND_DEV_CAP (continued)

Member Name	Value	Description
gain_caps	See Gain Capabilities Array	Pointer to SND_GAIN_CAP array
num_rates	30	Number of sample rates
sample_rates	See Sample Rates	Pointer to sample rate array
num_chan_info	1	Number of channel info entries
channel_info	See Number of Channels	Pointer to channel info array
num_cm	3	Number of coding methods
cm_info	See Encoding and Decoding Formats	Pointer to coding method array

Gain Capabilities Array

The following tables show the various gain capabilities for the Philips UCB1200. This information is pointed to by the `gain_cap` member of the `SND_DEV_CAP` data structure. See `SND_GAIN_CAP` for more information about this data structure. This driver allows control of following individual physical gain controls:

Table B-5 Individual Gain Controls

`SND LINE SPEAKER`	Output Attenuation
`SND LINE MIC`	Microphone Gain

The following tables detail the various individual gain capabilities:

Table B-6 Speaker Gain Enable

Member Name	Value	Step	HW	Level	Comments
lines	`SND_LINE_SPEAKER`	0-3	31	-69 dB	`default_level`
`sup_mute`	`TRUE`	4-7	30	-66.8 dB	
`default_type`	`SND_GAIN_CMD_MONO`	8-11	29	-64.7 dB	
`default_level`	`SND_LEVEL_MAX`	12-15	28	-62.5 dB	
`zero_level`	`SND_LEVEL_MIN`	...	...	...	
`num_steps`	`32`	112-115	3	-6.5 dB	
`step_size`	`216`	116-119	2	-4.3 dB	
`mindb`	`-6900`	120-123	1	-2.2 dB	
`maxdb`	`0`	124-127	0	0.0 dB	`zero_level`

Table B-7 Mic Gain Enable

Member Name	Value	Step	HW	Level	Comments
lines	SND_LINE_MIC	0-3	0	0 dB	zero_level
sup_mute	FALSE	4-7	1	0.7 dB	
default_type	SND_GAIN_CMD_MONO	...	...	...	...
default_level	SND_LEVEL_MAX	64-67	16	11.3 dB	default_level
zero_level	SND_LEVEL_MIN	...	...	...	...
num_steps	32	112-115		20.4 dB	
step_size	70	116-119	29	21.1 dB	
mindb	0	120-123	30	21.8 dB	
maxdb	2250	124-127	31	22.5 dB	

Sample Rates

Following is an abbreviated list of the supported sample rates for the UCB1200. Below is a formula to derive valid sample rates:

sample_rate = 11981000/(32 * i), where 8 < i < 128

This information is pointed to by the `sample_rates` member of the `SND_DEV_CAP` data structure.

Table B-8 Sample Rate (Hz)

2948	3941	4926	5942	6933
7966	8914	9852	10697	11700
12910	13866	14976	15600	17828
18720	19705	20800	22023	23400
24960	26743	28800	31200	34036
37440	41600	46801	53486	62401

Number of Channels

The following table shows the different supported number of channels for the Philips UCB1200. The first entry in the table is the default number of channels. This information is pointed to by the `channel_info` member of the `SND_DEV_CAP` data structure.

Table B-9 Number of Channels

Channels	Description
1	Mono

Encoding and Decoding Formats

The following table shows the supported encoding and decoding formats for the Philips UCB1200. The first entry in the table is the default format. This information is pointed to by the `cm_info` member of the `SND_DEV_CAP` data structure.

Table B-10 Encoding and Decoding Formats

Coding Method	Sample Size	Boundary Size	Description
`SND_CM_PCM_ULAW`	8	2	8 bit u-Law compounded
`SND_CM_PCM_SLINEAR` `SND_CM_LSBYTE1ST`	16	4	16 bit Linear (two's complement) little endian
`SND_CM_PCM_SLINEAR`	16	4	16 bit Linear signed (two's complement) big endian

SPUCB1200 driver for the UCB1200 Codec

This document describes the hardware specifications for the Philips UCB1200 driver. This is an SPF driver.

Capabilities

The UCB1200 is capable of controlling a microphone/speaker, input/output telecommunications lines, resistive style touch screen, and 16 General Purpose Input/Output lines. This driver currently can only control the touch screen, and general purpose input/output lines. The microphone/speaker can be controlled with a MAUI Sound driver called `sd_ucb1200`. No driver has been written for the telecommunications part of the UCB1200.

Descriptors

Table B-11 lists the UCB1200 descriptors.

Table B-11

Name	Function
`ucb`	UCB1200 Chip Initialization
`ucb_audio`	Not Implemented
`ucb_touch`	Touch Screen
`ucb_gpio`	Control GPIO Lines
`ucb_telecom`	Not Implemented

UCB

Opening the /ucb device will perform basic chip initialization. Normally this is not necessary, unless another driver is written to control part of the UCB1200 functions. This is the case for audio. The MAUI Sound driver `sd_ucb1200` will open /ucb to perform chip initialization. In this way, the MAUI Sound driver play audio and this driver can control the touch screen at the same time.

Audio

This portion of the driver is not implemented since the MAUI Sound driver `sd_ucb1200` already exists. `sd_ucb1200` and this driver can co-exist.

Touch Screen

This portion of the driver controls the touch screen operation. When pressure is applied to the touch screen, a hardware interrupt is raised, and this driver's interrupt service routine will execute. A system state alarm, then, will fire at regular intervals to sample data from the touch screen. When pressure is removed, the alarm stops. This mechanism leaves the UCB1200 in a low power state until the user presses the touch screen. The alarm rate can be controlled in the `ucb_touch` descriptor.

Each sample contains an x, y coordinate as well as pressure information. The data is formatted into a six byte packet as defined in the table below. Each packet contains 10 bits of x, 10 bits of y, and 8 bits of pressure information.

Table B-12 Touch Screen Descriptor Data

Byte number	Description
0	sync code - 0x80
1	header: bit 1: pendown bit 2: penup bit 3: penmove (may occur with pendown or penup)
2	bits 0..2: high 3 bits of x bits 3..6: high 4 bits of pressure bit 7: 0
3	bits 0..6: low 7 bits of x bit 7: 0
4	bits 0..2: high 3 bits of y bits 3..6: low 4 bits of pressure
5	bits 0..6: low bits of y bit 7: 0

GPIO

This section of the driver has basic GPIO line control, where lines 0..9 are connected to a 7 segment display or LED. Each line can be controlled with an `_os_write()` call. (Refer to the UCBHEX program in the TEST directory.)

Telecom

This portion of the driver is not implemented.

Supporting Modules

Before this driver can be used, the following modules must be in memory: `spf`, `sysmbuf`, `mbinstall`. `mbinstall` must also be run before use.

MP_UCB1200 MAUI Touch Screen Protocol Module

This section describes the function of the `mp_ucb1200` protocol module, as well as a high level discussion of the touch screen driver and calibration application.

Overview

The protocol module converts the driver raw data into a `MAUI_MSG` structure. In this way, applications can remain somewhat ignorant of the details of the hardware since it deals with the MAUI Input layer. In this protocol module, the raw hardware data is converted into screen coordinates. In addition, some data filtering occurs to reduce the amount of erroneous data that the touch screen hardware can produce.

Data Format

The touch screen driver sends a 6 byte packet that contains x, y, and pressure information. The exact format of this packet is described in the spucb1200 driver.

Data Filter

This protocol module filters the data coming from the hardware in an attempt to reduce erroneous data. Two methods are implemented: data point averaging and low pressure point removal. The first method will average the last two points received from the driver. The data point will lag slightly behind the current position, then, but the average will reduce erroneous data points produced by the hardware. The second method throw out data points where the pressure below a certain threshold. It seems that extremely light touches will cause the data to become erratic, although the exact pressure threshold is hardware dependent.

Raw Mode

An application can put this protocol module in a "raw" mode where data points are not filtered, averaged, or converted to screen coordinates. That is, the data from the hardware is passed directly up to the application.

The application can put this protocol module in a "raw" mode by calling: `inp_set_sim_meth(inpdev,RAW_MODE)`. After calibration, the program will need to put the protocol module back in NATIVE mode by calling: `inp_set_sim_meth(inpdev,DEFAULT_SIM_METH)`. There is a sample touch screen Calibration Application in the `TOUCH_CAL` directory.

When the protocol module is taken out of "raw" mode, it will try to read new calibration data points from the ucb1200.dat data module. After the data is read from the module, it is no longer needed.

cdb.touch

The touch screen can be registered with MAUI by loading the `cdb.touch` module in memory before any programs using input are started. This will specify the spucb1200 as the driver, `cdb.touch` as the descriptor, and `mp_ucb1200` as the protocol module.

Compile Time Options

Table B-13 shows compile time options used to control the default calibration settings and also the screen size. These options can be specified with a value in the `mp_ucb1200` makefile to modify the defaults.

Table B-13 Compile Time Options

Name	Purpose
`SCREEN_WIDTH`	Screen Width in Pixels
`SCREEN_HEIGHT`	Screen Weight in Pixels
`DEFAULT_CALIBRATION_X`	Left Calibration Hardware Point
`DEFAULT_CALIBRATION_Y`	Top Calibration Hardware Point
`DEFAULT_CALIBRATION_WIDTH`	Width of Screen In Hardware Points
`DEFAULT_CALIBRATION_HEIGHT`	Height of Screen In Hardware Points
`JITTER_THRESHOLD`	Minimum Pixel Change Required Before Points are Reported to the Application.
`NUM_PTS`	This allows you to choose how many successive data points to average in order to produce less erroneous screen coordinate data to the application. The default is 2, and valid choices are 1, 2, 4, 8, 16.
`MIN_PRESSURE`	Any pressure point less than this value will be ignored. This is another way to reduce erroneous data. This represents the 8 bit pressure value we get from the driver. The default is 40.

Calibration Application

There is a sample calibration application located in the `$(MWOS)/SRC/MAUI/MP/MP_UCB1200/TOUCH_CAL` directory. This application, called `touch_cal`, will present a text message on the screen as well as points for the user to press. After the points are pressed, the protocol module `mp_ucb1200` will be updated with the new calibration information.

Assumptions/Dependencies

1. A Window Manager must be running before this application will operate.
2. A font module must be present to run the demo. `default.fnt` is the default module, or you can specify one on the command line.
3. `touch_cal` will open the first `CDB_TYPE_REMOTE` device in the `cdb`.

Command Line Options

`-f[=]<outfile>`	Specifies the filename of the calibration information module. This program will write the calibration information to this filename if it is specified. The file contains the calibration information as a data module, thus allowing the information to be stored on disk, nv RAM, flash, etc. for use the next time the hardware is rebooted.
`-c`	This option only works if `-f` is specified. This will cause the calibration program to run only if the filename specified with `-f` is not present.
`-m=<font module>`	Specifies the font module to use for displaying the text message on the screen.

Coordination with Protocol Module

The protocol module `mp_ucb1200` and the touch screen application `touch_cal` work together to provide the calibration functionality. `touch_cal` must first open the touch screen device, and then must set it into Raw Mode. After the user selects each calibration point, `touch_cal` computes the average of them. These averaged hardware points (as well as the screen resolution) are then stored in a data module called `ucb1200.dat.` When the input device is taken out of Raw Mode, the protocol module will link to `ucb1200.dat` and update itself with the new calibration information.

Compiling

The makefile for `touch_cal` exists in the `$(PORTS)//MAUI/MP_UCB1200/TOUCH_CAL` directory.