NIOS II 软件开发手册

编辑：DiDy

2006-11-29

Email:wedidy@gmail.com

目录

第一部分 Nios II 软件的发展

这一部分主要介绍 Nios II 软件发展的相关信息。包括以下两章：
第一章　概述
第二章　Nios II IDE 简介

第一章 概述

1.1 引言

这一章从较高的角度为 Nios II 软件开发人员总体介绍 Nios II 处理器。对 Nios II 软件的开发环境，用户可以使用的工具和相应的设计过程进行讲解。

1.2 Nios II 简介

基于 Nios II 处理器同使用其他微处理器的软件设计相似。最简单有效的方法是从 Altera 公司购买一套开发工具，其中包括相应的文档，开发板和编写 Nios II 程序相应的开发工具。

Nios II 软件开发者手册是针对那些对 Nios II 处理器有一定了解的用户。使用者不需要精通 Altera 的技术和相应开发的工具。精通 Altera 硬件开发工具的用户可以更深入的理解 Nios II 软件开发环境。但对于了解 Altera 开发工具较少的用户，同样也可以使用其开发和调试。

在学习新的软件开发环境的时候，大多用户都习惯在原有程序基础上进行修改。Nios II 开发套件提供了大量的设计例程，以方便用户在其基础上进行检查，修改或者直接应用到设计中。提供的例程从简单的“Hello Word”到复杂的实时操作系统，基于 TCP/IP 协议的网络服务器。每一个例程均备有文档说明，并可以编译。

1.3 开发环境

这节主要介绍 Nios II 软件开发环境。

1.3.1 工具：

Altera Nios II 处理器提供下列使用工具
1）Nios II IDE
2）GNU 工具链
3）指令设置仿真
4）硬件提取层系统库

5）RTOS 和 TCP/IP 堆栈
6）例程

Nios II IDE

Nios II 完整的开发环境（IDE）是 Nios II 处理器与用户的接口界面。使用 Nios II IDE 可以完成所有的软件开发任务。包括程序的编辑，编译和调试，它是其他工具的一个窗口。

GNU 工具链

Nios II 编译器是基于标准 GNU GCC 基础上进行，编译，汇编，连接使用的。

指令设置仿真

Nios II 指令设置仿真（ISS）可以使用户在硬件平台没有完善的情况下，对系统进行程序的编写和仿真，Nios II IDE 使用户在 ISS 上运行程序与在真实的硬件上运行一样简单轻松。

硬件提取层系统库

硬件提取层（HAL）支持通用 IO 器件，可以通过编写标准 C 程序访问硬件。如 printf( )，HAL 减少了对硬件寄存器的访问，直接与外围器件进行通信或控制。

不同的外围硬件需要不同程度的 HAL 的支持，需要运行 HAL 的软件驱动器。如需要更多的了解 HAL，可以参见本手册的 *HAL 系统库*和 *HAL 应用程序接口参考*。

RTOS 和 TCP/IP 堆栈

Altera 支持 MicroC/OS-II RTOS 和简化 TCP/IP 堆栈。MicroC/OS-II 是基于 HAL 系统库上，执行简单的，备有证明文件的 RTOS 调度程序。TCP/IP 堆栈是基于 MicroC/OS-II 的，执行标准的 UNIX Sockets 应用程序接口。其他的操作系统和堆栈设置可以通过第三方获得。

例程

备有许可证的软件例程可以为所有 Nios II

统一的开发环境

Nios II IDE 为 Nios II 处理器提供统一开发环境。如果你拥有一台 PC 机，一个 Altera 的 FPGA 和一个 JTAG 下载线，就可以任意的 Nios II 处理器相连接。JTAG 下载电线可以提供单步和连续运行。

统一的运行时间环境

系统 HAL 系统库包含了统一的，主机 C/C++运行时间环境，而不用考虑底层硬件系统。开发包中自带的可以自动生成自定义 HAL 系统库，所以用户不需要手动去写驱动和开发办支持包。

用户可以很容易的减少 HAL 的运行时间，来缩减系统的代码量。也可以使用独立的 C 环境，来完成全部的控制，以及系统初始化和底层硬件驱动。

1.4 第三方支持

一些支持 Nios II 的第三方厂家，提供一些如设计服务，RTOS，其他软件库或者一些开发工具。

如果想了解更多有关 Nios II 第三方支持，可以登录 Altera 的网站 www.altera.com/nios2。

1.5 第一代 Nios 处理器的移植

如果用户使用的是第一代的 Nios 处理器，Altera 建议在以后的设计中将第一代移植到第二代处理器 Nios II 上。在开发文档 AN350 上介绍有关移植的方法。

第二章 Niso II IDE

引言

这部分主要使用户熟悉 Nios II 综合开发环境（IDE）。这一部分只是简要介绍一下 Nios II IDE 的界面和使用——并不是用户指导。开始使用 Nios II IDE 的最简单方法，是依照系统帮助中的 Nios II 软件开发指南运行 Nios II IDE。

Nios II IDE 工作台

这里所谓的“工作台”指的是 Nios II IDE 开发系统提供的界面。在工作台上编写，编译和调试程序。图 2.1 为 Nios II IDE 工作台

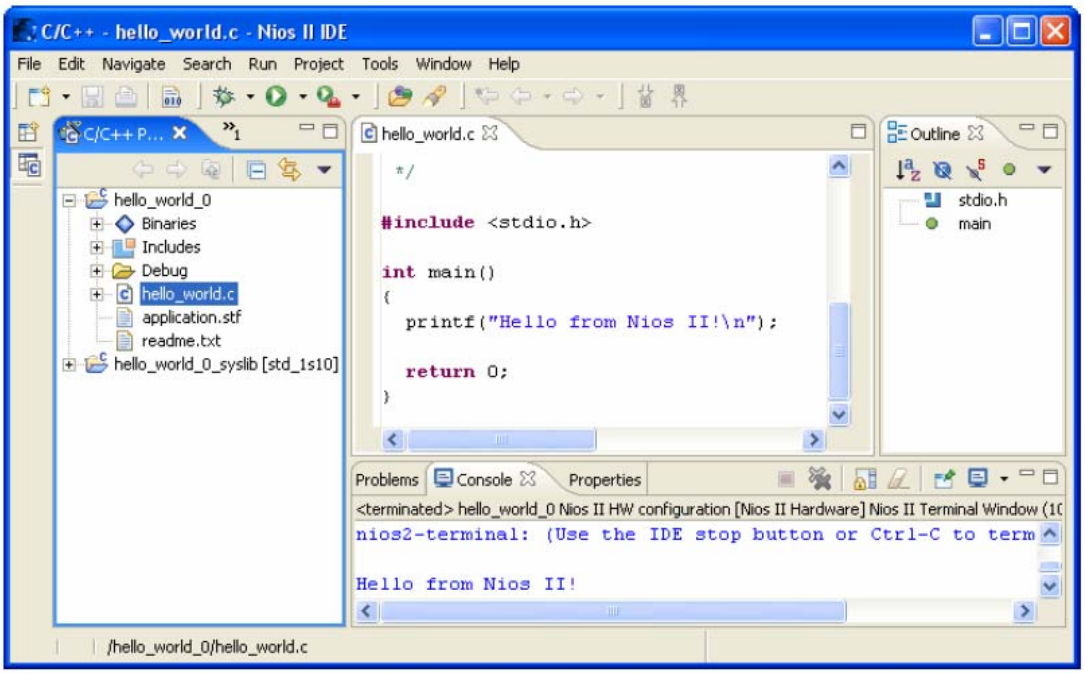

图 2.1 Nios II IDE 工作平台

建立新的工程

Nios II IDE 提供新工程向导，指导用户按步骤建立新 C 或 C++工程。要建立新的 C/C++应用，点击 File，New，C/C++ Application。如图 2.2

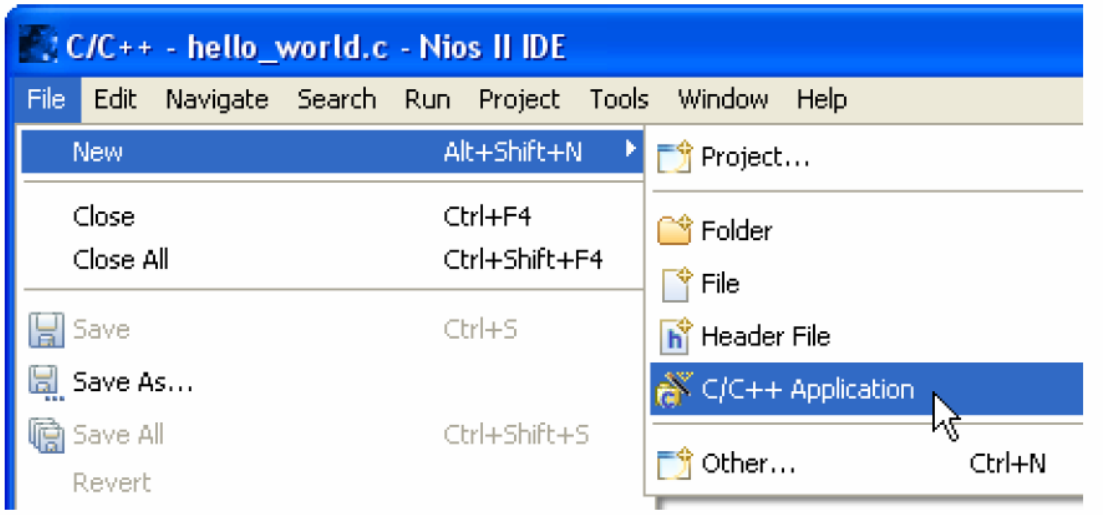

图 2.2 建立新工程向导

C/C++ 新工程向导的提示如下：

1．新工程的名字

2．目标硬件

3．新工程的模版

工程模版已经为用户设计完成，为用户构建工程提供向导服务。开始可以从简单的“Hello Word”入手进行学习设计，然后再在进行空白工程设计。

如图 2.3 所示的 C/C++新工程向导，以 Dhrystone 为模版进行设计。

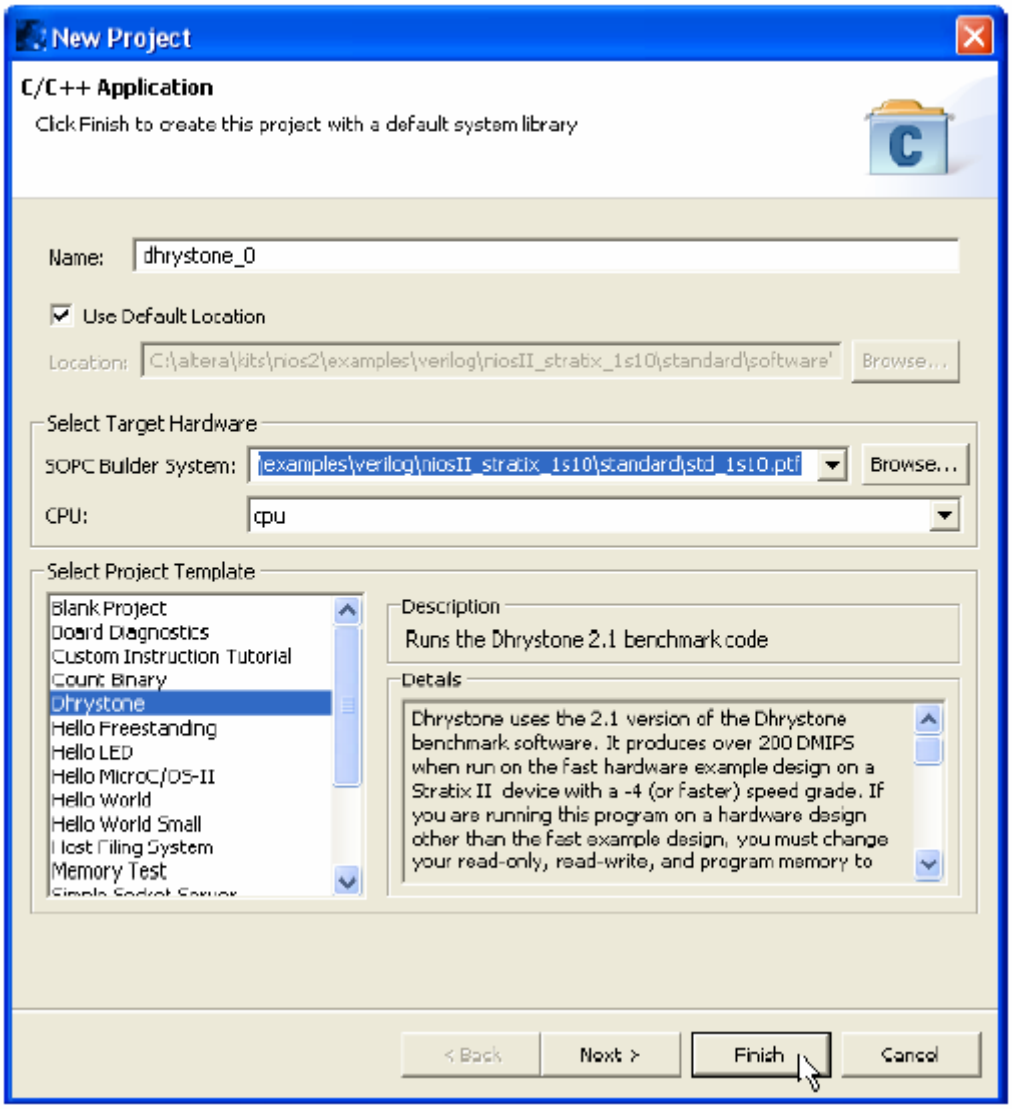

图 2.3 C/C++ 新工程向导

当点击 Finish，Nios II IDE 完成新工程的建立。IDE 同样创建系统库工程。

组建并编辑工程：

右键单击任何资源（如，文件，文件夹或者工程）会出现相应操作的选项。右键单击通常会出现最常用的选项。

要编译工程，可以右键单击 C/C++ Projects 窗口，然后选择 Build Project。图 2.4 为在 dhrystone_0 工程的下拉菜单中选中 Build Project 选项。在构建工程时，Nios II IDE 首先会构建系统库，然后编译主要工程，所有的警告和错误会显示在 Tasks 窗口中。

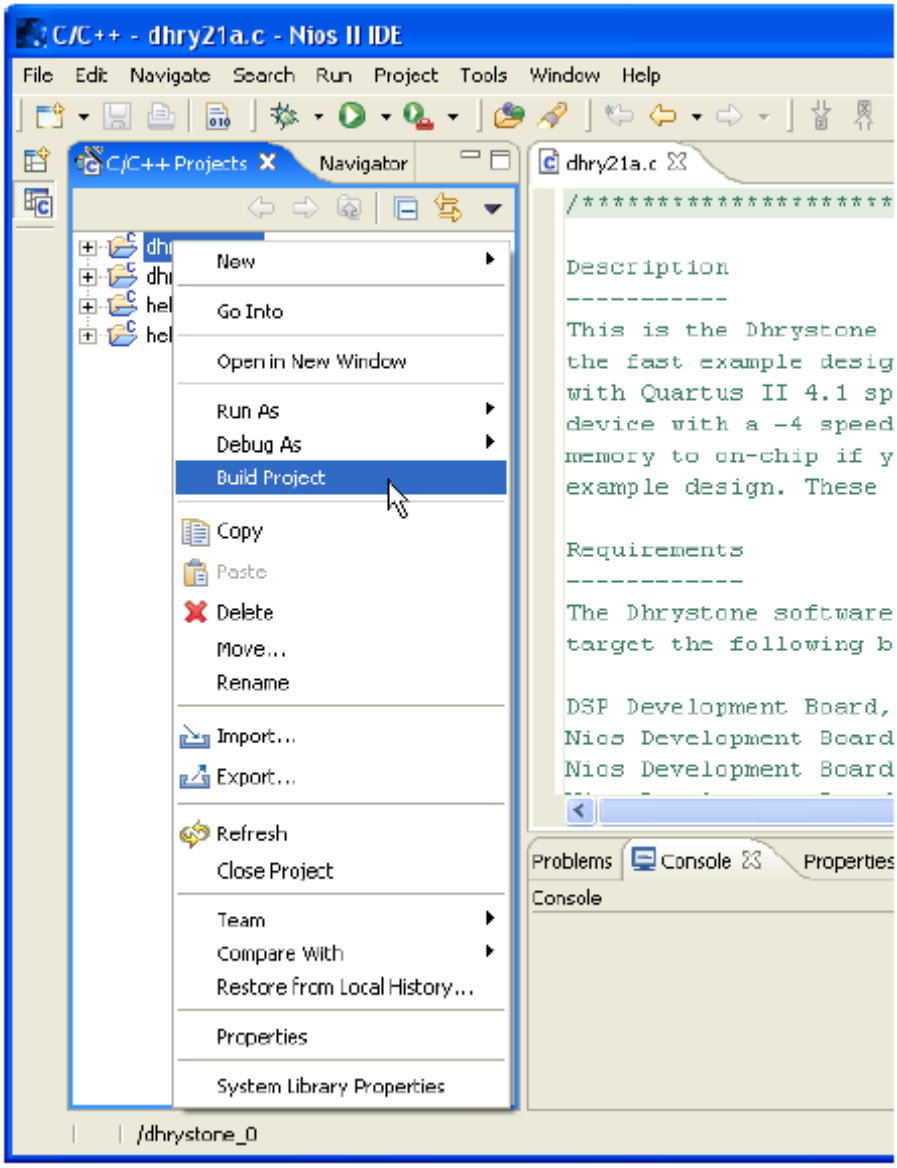

图 2.4 构建工程

右键单击 C/C++工程还可以完成如下操作，对工程进行管理。

➔ Properties 对于目标硬件或工程相关资源管理。

➔ System Library Properties 管理硬件特性设置，如通信设置和存储器分配。

➔ Build Project 构建工程

➔ Rebuild Project 全部构建

➔ Run as 在硬件或者 ISS 上运行

➔ Debug as 在硬件或者 ISS 上调试

运行和调试程序

单击右键可以选择运行（Run）或者调试（Debug）选项。Nios II IDE 可以使用户在硬件上运行，调试程序，也可以通过指令集（ISS）仿真运行和调试。例如，要在硬件上运行程序，在 C/C++ Projects 窗口中单击右键，在出现的下拉菜单中单击 Run As **Nios II Hardware** 如图 2.5

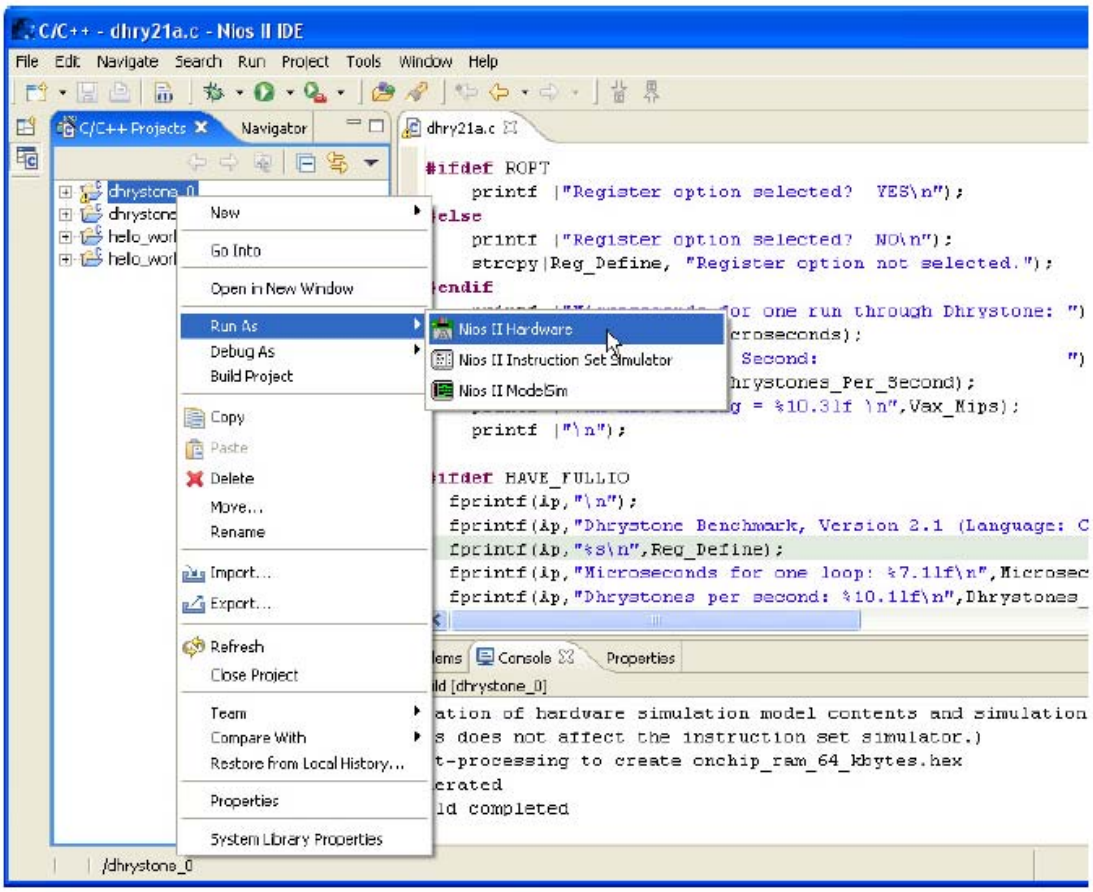

图 2.5 Run As

如图 2.6 在 ISS 上调试程序

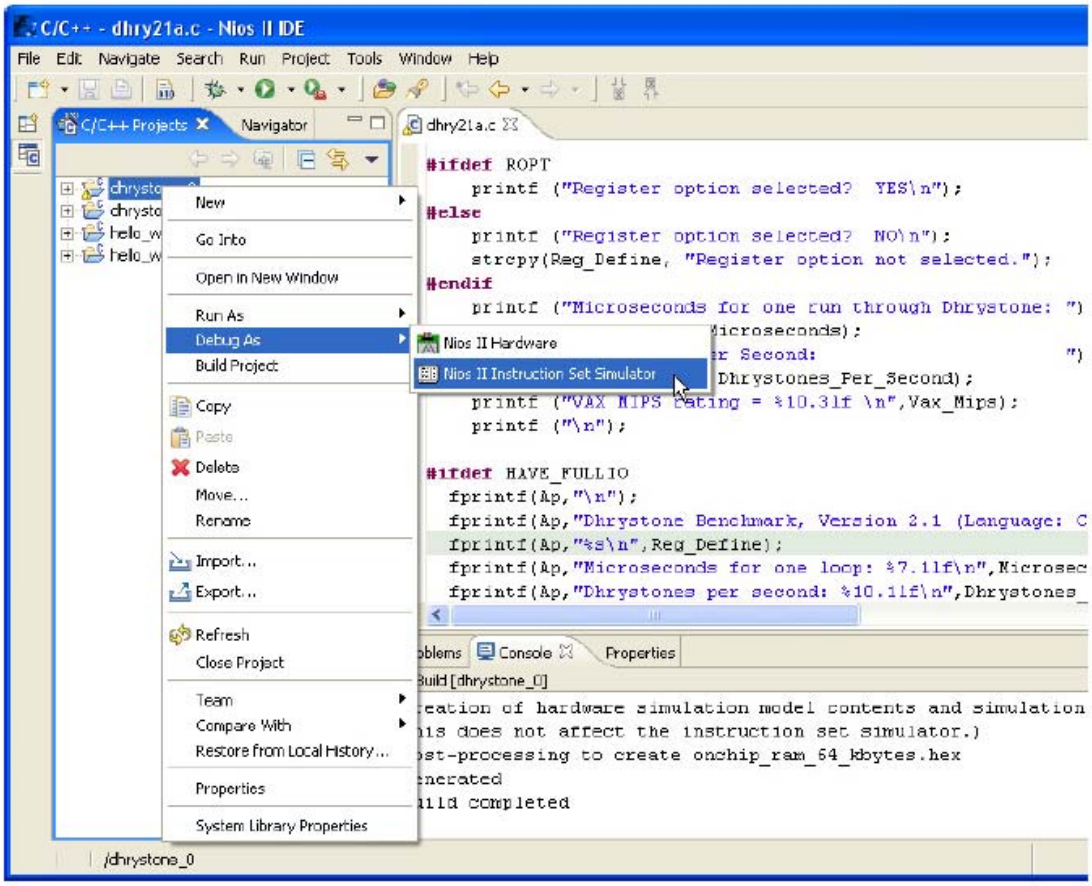

图 2.6 Debug As

图 2.7 为进入调试模式时的界面。

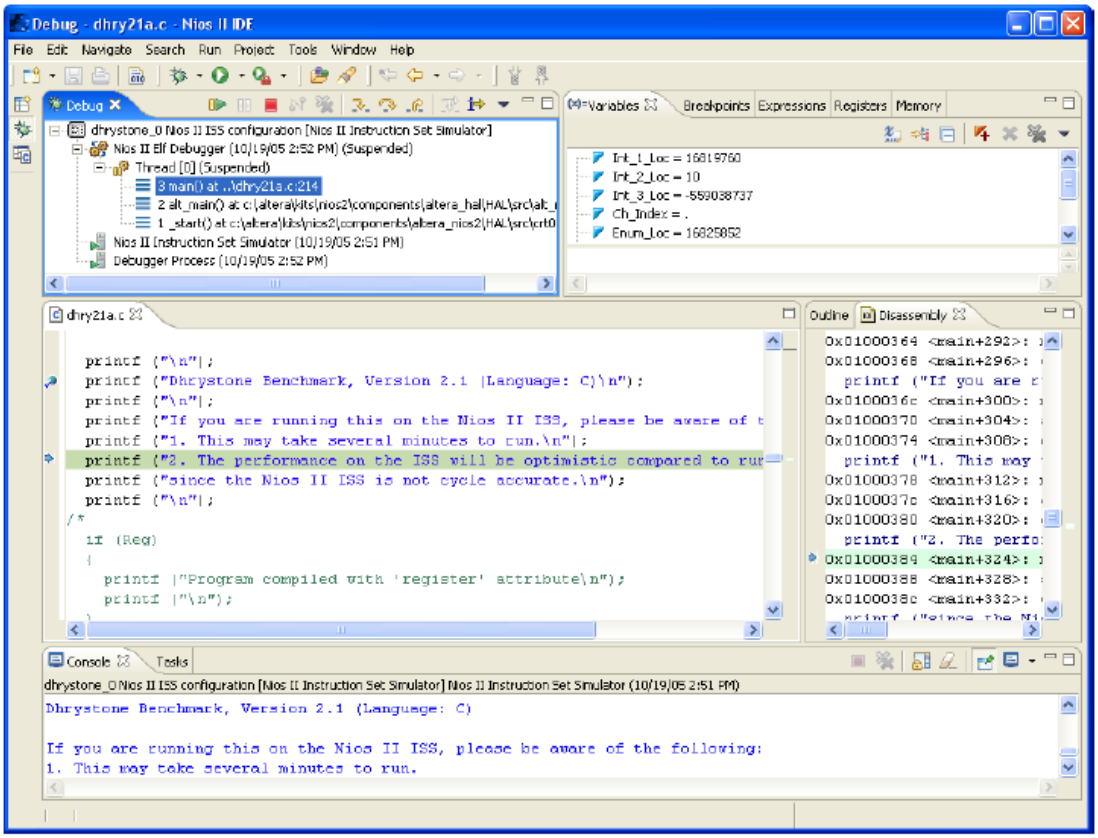

图 2.7 调试界面

进行调试时，软件转换成调试界面。用户可以点击左边工作窗中的图标，简单的在 Debuger 界面和 C/C++ Projects 界面之间切换。

当进入调试对话框时，调试器载入程序，在主函数 main( ) 中设置断点，然后开始运行程序。用户可以全速运行，单步等操作。在代码左端双击设置断点，或者右键选择断点选项。

Nios II IDE 提供很多调试窗口，可以使用户方便的观察运行时处理器的状态。如，变量，公式，寄存器，存储器等等，如图 2.8 所示的寄存器窗口。

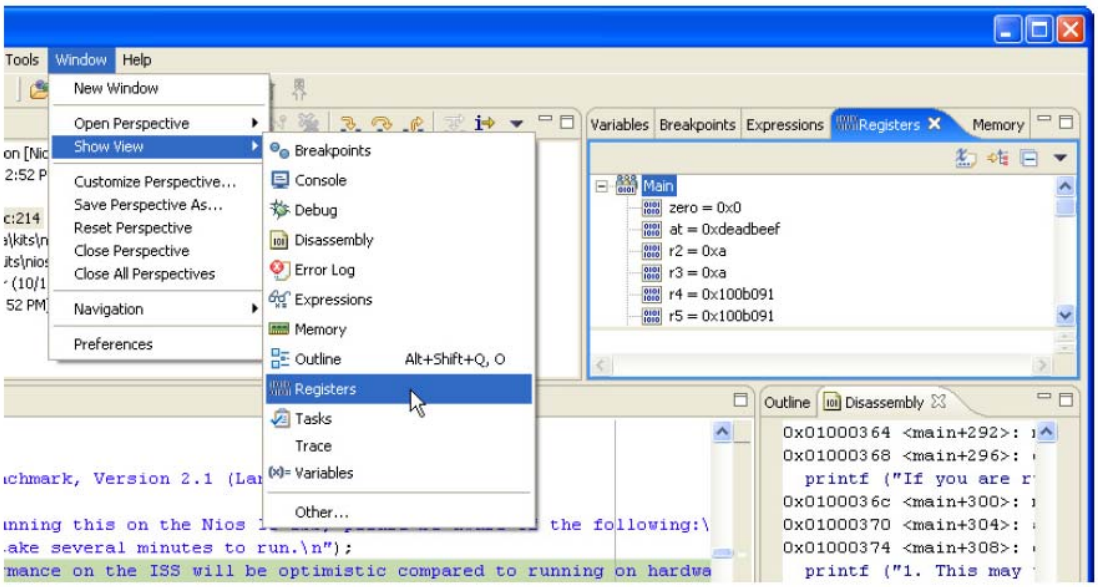

图 2.8 寄存器窗口

编辑 Flash

很多 Nios II 处理器使用外部存储器来存储如下一项或多项内容：

➔ 程序代码

➔ 程序数据

➔ FPGA 配置信息

➔ 文件系统

Nios II IDE 提供 Flash 编辑器程序，帮助用户管理 Flash，对其进行写入或擦除。如图 2.9 为 Flash 编辑器。

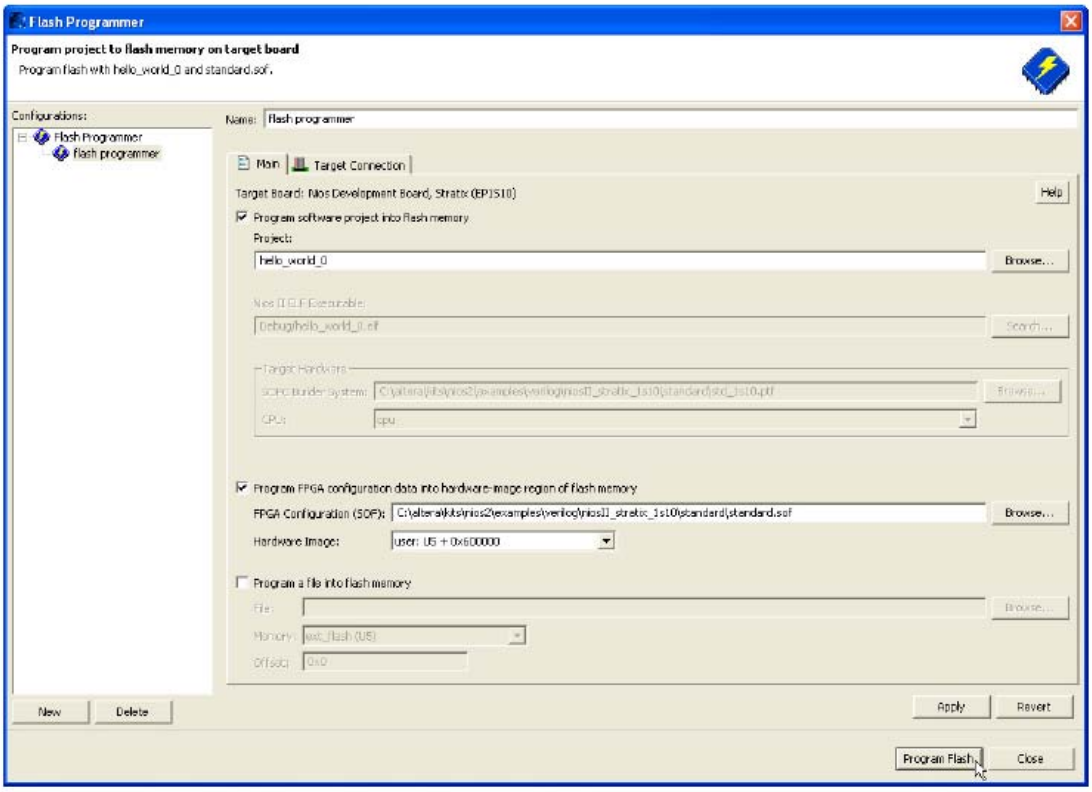

图 2.9 Flash 编辑器

帮助系统

Nios II IDE 帮助系统提供最新的文档。点击 Help，如在 Windows 系统中可以按 F1 键。帮助文档可以帮助用户按步骤进行学习操作——建立工程，构建工程，调试等等。如图 2.10 所示的帮助选项对话框。

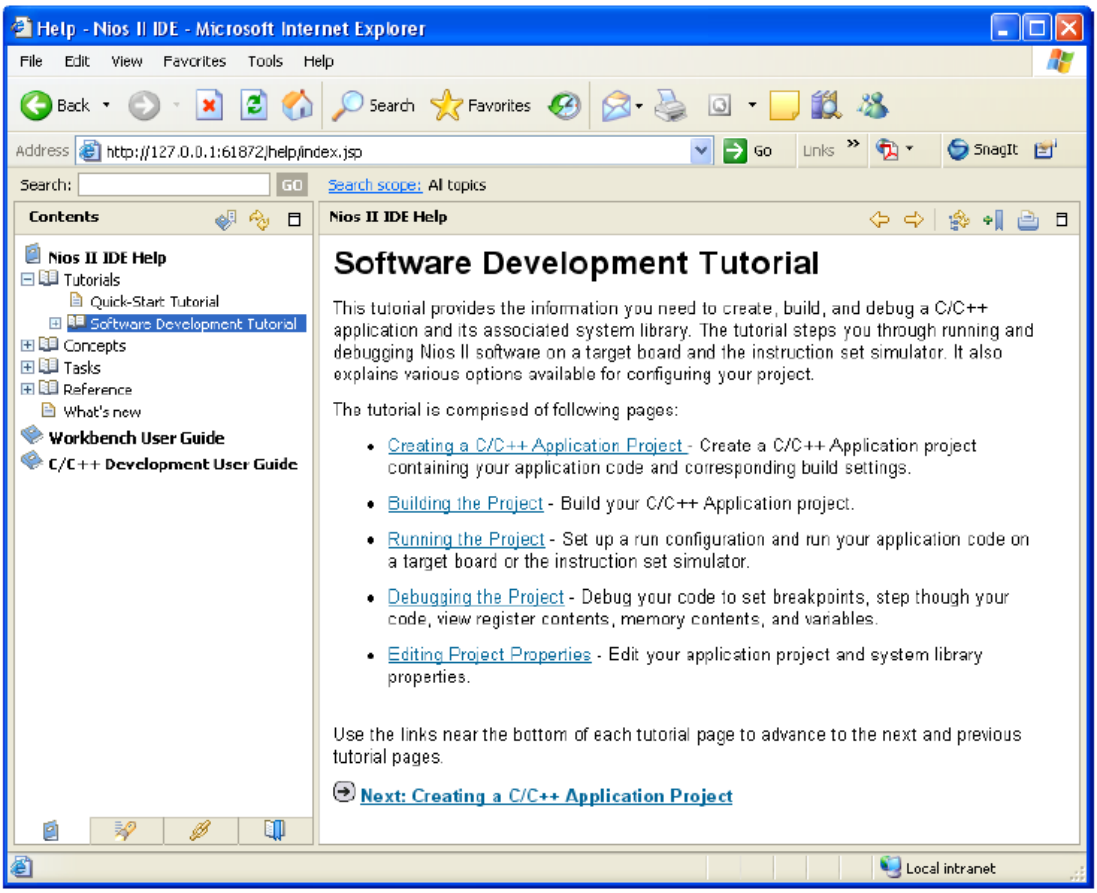
Help - Nios II IDE - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:61872/help/index.jsp
Go Links SnagIt
Search: GO Search scope: All topics
Contents
Nios II IDE Help
Tutorials
Quick-Start Tutorial
Software Development Tutorial
Concepts
Tasks
Reference
What's new
Workbench User Guide
C/C++ Development User Guide
Nios II IDE Help
Software Development Tutorial
This tutorial provides the information you need to create, build, and debug a C/C++ application and its associated system library. The tutorial steps you through running and debugging Nios II software on a target board and the instruction set simulator. It also explains various options available for configuring your project.
The tutorial is comprised of following pages:
Creating a C/C++ Application Project - Create a C/C++ Application project containing your application code and corresponding build settings.
Building the Project - Build your C/C++ Application project.
Running the Project - Set up a run configuration and run your application code on a target board or the instruction set simulator.
Debugging the Project - Debug your code to set breakpoints, step though your code, view register contents, memory contents, and variables.
Editing Project Properties - Edit your application project and system library properties.
Use the links near the bottom of each tutorial page to advance to the next and previous tutorial pages.
Next: Creating a C/C++ Application Project
Local intranet

图 2.10 帮助对话框

第二部分 HAL 系统库

第三章 HAL 系统库简介

引言

这一章主要介绍 Nios II 处理器的硬件抽象层系统库。HAL 系统库是一个底层的运行环境，为系统提供底层硬件的驱动。HAL 系统库应用程序接口（API）由标准的 ANSI C 库组成。HAL 的 API 允许用户使用熟悉的 C 语言的函数对器件进行访问，如 printf( )，fopen( )，fwrite( )等等。
HAL 作为 Nios II 处理器的开发板套件，为嵌入式系统的外围器件提供接口，将 Altera 的 SOPC Builder 和 Nios II IDE 紧密的联系在一起，自动生成 HAL 系统库。在 SOPC Builder 生成系统硬件后，Nios II IDE 创建生成 HAL 系统库以匹配生成的硬件系统。当系统硬件配置发生改变时，系统自动将信息传给 HAL 系统库，以减少由于底层硬件发生微小变化而产生的系统错误。
HAL 系统库对具体应用和器件驱动软件设计的支持有明显的分别。在硬件不改变的情况下可以重复使用系统提供的应用代码。另外，编写与现有支持的外围器件相兼容的外围器件的驱动也非常简单。

开始设计

在 Nios II IDE 中建立新的工程的过程中，系统自动生成 HAL 系统库。您不需要创建或者复制 HAL 文件，同样也不需要对任何 HAL 源代码进行修改。Nios II IDE 自动创建 HAL 系统库，同时对其进行管理。
HAL 系统库建立在特定的 SOPC Builder 系统之上。综合外围器件和存储器，SOPC Builder 生成完整的 Nios II 处理器。如果您没有建立自己的 SOPC Builder 系统，可以使用 Altera 提供的硬件系统例程。事实上，用户可以先采用 Altera 提供的 Nios II 系统开发板进行设计，然后再将其移植到您自己的硬件系统上，这会使设计变得简单。

HAL 的结构

这一节介绍 HAL 体系结构的基本单元。
支持服务
HAL 系统库支持如下服务：
➔综合最新的 ANSI C 标准库：支持用户熟悉的标准 C 函数库
➔器件驱动器：支持访问每一个器件
➔HAL API：支持兼容，标准接口。如器件访问，中断处理和报警设备等等
➔系统初始化：在 main( )函数运行前，对处理器进行初始化
➔器件初始化：在 main( )函数运行前，对各个器件进行初始化

如图 3.1 给出了 HAL 基本结构层设置，从底层的硬件层到用户的程序层。

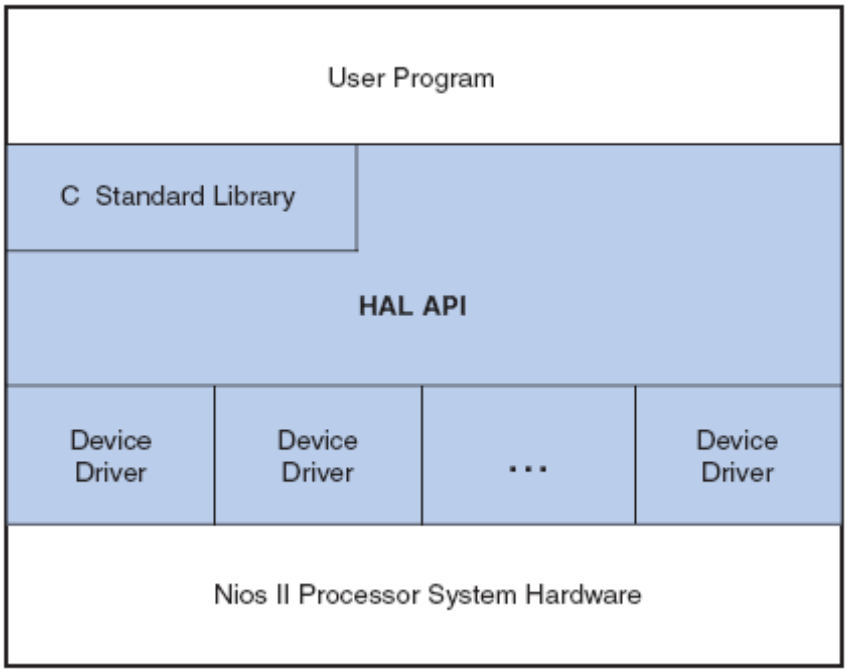

图 3.1 HAL 基本系统结构层

应用设计与驱动设计：

程序师明显被划分为两组：应用开发人员和设备驱动开发人员。大多数用户为应用开发设计者，一般负责系统得 main( )程序。应用设计一般使用标准的 C 函数库或者 HAL 系统库 API，进行系统资源调用。设备驱动开发人员一般设计设备驱动供应用开发使用。设备驱动开发人员直接对系统底层硬件进行操作处理。

通用器件模板

HAL 为嵌入式系统外围器件体统通用的模板。如：定时器，以太网 MAC/PHY 芯片以及按字节传输的 I/O 外围设备。外围器件通用模板可以使您使用通用的 API，而不用考虑底层硬件。

器件模板种类

HAL 为下列器件提供模板：

➔字符模式器件——串行收发字节的外围器件，如 UART

➔定时器器件——可以对时钟信号进行计数，并产生周期中断的外围器件

➔文件子系统——支持访问文件存储，由于是内部执行操作，文件子系统驱动器可以直接访问底层硬件，例如：您可以编写 Flash 文件子系统驱动，通过 HAL 的 API 接口来访问 Flash。

➔以太网器件——使 Altera 的低标准 IP 协议接入以太网接口

➔DMA 器件——支持大量数据高速传输

➔Flash 存储器件——用于存储数据或程序的非易失性存储器

对应用开发工程师有利的方面

HAL 系统库定义一组基本功能，使用户可以对常用器件进行初始化和访问。通用的 API 接口，使用户不需要考虑底层硬件。例如：如要使用 UART，或是文件子系统，用户可以使用标准 C 语言函数，如 printf( )和 fopen( )。所以对于应用开发工程师来讲，不需要自己编写底层硬件的驱动。

对器件驱动工程师有利的方面

每一个器件模式都定义了控制特殊器件的基本功能。如果您要为一个新的外围器件写驱动程序，只需要写驱动功能。另外，现行的 HAL 功能和应用可以对器件进行访问，节省开发时间。HAL 系统库调用驱动程序访问硬件。应用工程师使用 ANSI C 和 API 访问器件，而不是直接调用驱动程序。因此用户编写的驱动是作为系统 HAL API 的一部分。

C 标准库

HAL 系统库将 ANSI C 综合到其运行环境。HAL 使用 newlib，是 C 标准库的开源实现。Newlib 特别适合 HAL 和 Nios II 处理器的嵌入式系统的 C 语言库。Newlib 是免费使用的。

支持外围设备

Altera 的 Nios II 处理器支持很多外围器件。大多数外围器件支持 HAL 器件驱动，允许用户通过 HAL 的 API 接口对底层硬件进行访问。HAL 支持以下器件。

■ 字符模式器件:
● UART核
● JTAG UART 核
● LCD 16207显示控制器
■ Flash存储器件
● 通用flash接口的flash器件
● Altera’s EPCS 串行配置芯片控制器
■ 文件子系统
● 只读文件系统
■ 定时器器件
● 定时器内核
■ DMA 器件
● DMA 控制核
■ 以太网器件
● LAN91C111以太网MAC/PHY控制器

注：LAN91C111 器件需要 MicroC/OS-II 运行环境

所有的外围器件必须提供定义其底层硬件接口的头文件。由此看来，在某种程度上外围器件支持 HAL。但是有些器件没有提供器件驱动。如果驱动不可用，则只有通过器件的头文件中给的说明去访问器件。

不可避免的有一些器件对接口有特殊要求，所以不能使用通用的 API。HAL 系统库对于特殊器件，使用 UNIX 模式中的 ioct1( )功能。因为硬件是建立在外围器件上的，而 ioct1( )操作具有对每一个外围器件都有相应的说明。

有些器件提供专门的访问功能，而不是建立在 HAL 通用器件模式之上的。例如：Altera 为 Nios II 嵌入式处理器提供通用的并行 I/O（PIO）内核。PIO 不完全支持 HAL 通用器件模式中的所有器件，所以它只给出一个头文件和一些专用存取程序功能。

DiDy
2006-10-7
Email:wedidy@gmail.com

第四章 使用 HAL 进行程序开发

引言：

这一章主要讨论怎样在 Altera 的硬件抽象层（HAL）系统库基础上编写开发程序。对于刚接触 Nios II 处理器的软件开发者来说，可以访问以 HAL 为基础 API。使用 ANSI C 标准库功能和运行环境编写的程序建立在 HAL 之上，通过 HAL API 的通用器件模式对硬件进行访问。HAL API 在很大程度上与熟悉的 ANSI C 标准功能保持一致，尽管 ANSI C 标准库与 HAL 系统库相互分离。将 ANSI C 和 HAL 紧密结合在一起开发相应的程序，但 HAL 不能直接调用。例如：用户可以使用 printf( ),scanf( )等 ANSI C 标准库提供的 I/O 函数，对字符模式器件进行访问。

Nios II IDE 工程结构

创建和管理基于 HAL 系统库的软件工程与 Nios II IDE 联系紧密。为理解 HAL 系统库，本节重点讨论作为 HAL 基础的 Nios II IDE 工程。
图 4.1 所示的 Nios II 程序框架，重点强调了 HAL 系统库是如何与之配合。标注显示模块的由来。

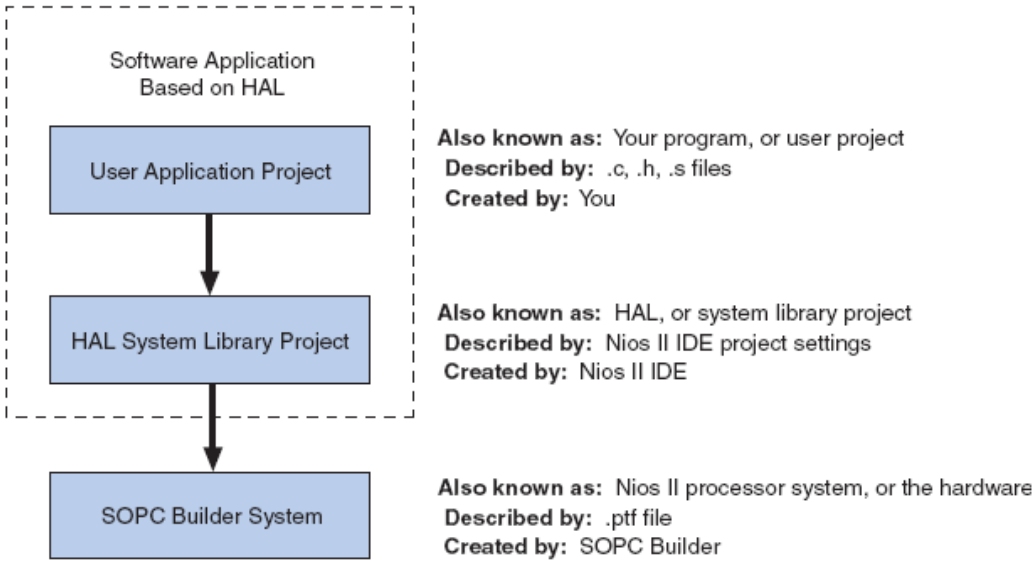

图 4.1 Nios II 程序框架

基于 HAL 的 Nios II 程序有两个 Nios II IDE 工程组成，如图 4.1 所示。用户程序在一个工程之中（User Application Project），并且建立在一个分离的系统库之上（HAL System Library）。用户的代码都包括在应用工程中（Application Project）。对这个工程进行编译得到最终的运行程序。
当用户建立应用工程时，系统建立 HAL 系统库。HAL 包含了所有与底层硬件与用户程序接口的有关信息。编译时，与用户 SOPC 系统相关的 HAL 驱动自动被添加到系统库工程中。HAL 的设置保存为系统性质。

建立在 SOPC Builder 之上的系统库工程，由 SOPC Builder 生成的**.ptf** 文件定义。Nios II IDE 管理 HAL 系统库和驱动配置的更新来正确反映硬件系统。当 SOPC 系统发生改变，IDE 将在用户编译和运行应用程序时将重新对 HAL 进行编译。这种工程从属结构显示将用户程序与底层硬件相分离，使用户不用担心程序是否与目标硬件相匹配。总之，基于 HAL 系统库的程序始终与硬件系统保持同步。

System.h 系统说明文件

System.h 是 HAL 系统库的基础。System.h 文件为 Nios II 硬件系统提供完整的软件说明，是硬件与软件的连接点。对于编程人员来讲，并非 system.h 提供的所有信息都有用。不需要明确的将其包括到用户的 C 源程序中。但是，如果想知道当前系统所使用的是什么硬件，在 system.h 文件中可以找到相应答案。
System.h 文件对系统中的每一个外围设备进行了说明，并提供如下的详细信息：
1．外围设备的硬件配置
2．基地址
3．中断优先级
4．外围设备符号名称
Nios II IDE 为 HAL 系统库生成 system.h 文件。System.h 的内容由硬件配置和用户在 Nios II IDE 中设置的 HAL 系统库性质决定。下面的一部分代码是 system.h 中的一部分，给出了一些硬件的配置信息。
例如：从 system.h 中摘取部分

```
/*
* sys_clk_timer configuration
*
*/
#define SYS_CLK_TIMER_NAME "/dev/sys_clk_timer"
#define SYS_CLK_TIMER_TYPE "altera_avalon_timer"
#define SYS_CLK_TIMER_BASE 0x00920800
#define SYS_CLK_TIMER_IRQ 0
#define SYS_CLK_TIMER_ALWAYS_RUN 0
#define SYS_CLK_TIMER_FIXED_PERIOD 0
/*
* jtag_uart configuration
*
*/
#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00920820
#define JTAG_UART_IRQ 1
```

数据宽度和 HAL 类型定义

对于与 Nios II 处理器相似的嵌入式处理器，了解数据的宽度和精度非常重要。

因为 ANSI C 并没有明确定义数据宽度，所以 HAL 使用一组标准类型定义。ANSI C 支持这种数据类型，数据宽度由编译器设定。
alt_types.h 头文件定义了 HAL 类型定义。如表 4.1 列出了 HAL 类型定义。
表 4.1

Table 4–1. The HAL Type Definitions

Type	Meaning
alt_8	Signed 8-bit integer.
alt_u8	Unsigned 8-bit integer.
alt_16	Signed 16-bit integer.
alt_u16	Unsigned 16-bit integer.
alt_32	Signed 32-bit integer.
alt_u32	Unsigned 32-bit integer.
alt_64	Signed 64-bit integer.
alt_u64	Unsigned 64-bit integer.

表 4.2 列出了 Altera 支持的 GNU 工具所使用的数据宽度
表 4.2

Table 4–2. GNU Toolchain Data Widths

Type	Meaning
char	8 bits.
short	16 bits.
long	32 bits.
int	32 bits.

UNIX 风格的接口

HAL API 提供很多 UNIX 风格的功能。UNIX 风格的工程为刚入门程序员提供了熟悉的编辑环境，简化了将代码移植到 HAL 环境下运行的步骤。HAL 利用这些工程首先为 ANSI C 标准库提供系统接口。例如：在 C 库函数中，对器件的访问功能都定义在 stdio.h 中。下面列出了所有的 UNXI 风格的功能函数。

- _exit( )
- close( )
- fstat( )
- getpid( )
- gettimeofday( )
- ioctl( )

- isatty( )
- kill( )
- lseek( )
- open( )
- read( )
- sbrk( )
- settimeofday( )
- stat( )
- usleep( )
- wait( )
- write( )

一般常用到与 I/O 文件关联的功能。

文件系统

HAL 为 UNIX 风格的文件的访问提供了基础结构。用户可以利用这个基础结构在用户硬件中使用的存储器件上建立文件系统。

用户既可以通过建立在以 HAL 为基础的系统库的 C 标准 I/O 函数对文件进行访问，也可以通过 HAL 系统库支持的 UNXI 风格的 I/O 功能。如下列出 UNIX 风格的函数：

- close( )
- fstat( )
- ioctl( )
- isatty( )
- lseek( )
- open( )
- read( )
- stat( )
- write( )

这里没有当前目录的概念，软件只能使用绝对路径访问文件。

HAL 文件基本结构支持用户通过 UNIX 风格路径名访问字符型器件。HAL 将字符型的器件视为 HAL 文件系统中的节点。按规定，system.h 定义这个字节型器件为在 dev/后面，以 SOPC Builder 分配给器件的名字，例如，对于一个 UART 外围器件 uart1 在 SOPC 将其建立在 system.h 中 dev/uart1。

下面的代码给出了读取只读 zip 文件系统 rozipfs 读取字节。

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#define BUF_SIZE (10)
int main(void)
{
FILE* fp;
char buffer[BUF_SIZE];
fp = fopen (“/mount/rozipfs/test”, “r”);
```

```
if (fp == NULL)
{
printf (“Cannot open file.\n”);
exit (1);
}
fread (buffer, BUF_SIZE, 1, fp);
fclose (fp);
return 0;
}
```
从例程可以看出其读取方式与 C 语言基本相同。

使用字符模式器件

这里所说的字符式器件指的是串行发送和接收字节的硬件外围设备。最常见的是通用异步接收发射（UART）。字符模式器件被作为 HAL 文件系统的节点。HAL 同样支持调用 stdio.h 标准输入输出功能。

标准输入输出和错误

使用标准输入（stdin），标准输出（stdout）和标准误差（stderr），是控制 I/O 的最简单方法。HAL 系统库在后台管理 stdin，stdout 和 stderr，这样使得用户在不用担心文件描述的处理的情况下，进行收发字节。例如，系统直接使用 printf( ) 作为标准输出。

下面这段代码是经典的 Hello Word。程序通过在 Nios II IDE 中编译时与系统连接的 stdout 输出字节。

```
#include <stdio.h>
int main ()
{
    printf (“Hello world!”);
    return 0;
}
```

访问字符模式器件

访问字符模式器件（包括 stdin，stdout，stderr）与打开、写入文件一样简单。下面的例子说明了怎样对 UART 写信息。

```
#include <stdio.h>
#include <string.h>
int main (void)
{
    char* msg = “hello world”;
    FILE* fp;
    fp = fopen (“/dev/uart1”, “w”);
    if (fp!=NULL)
    {
        fprintf(fp, “%s”,msg);
        fclose (fp);
    }
```

```
    return 0;
}
```

使用定时器器件

定时器是对时钟信号进行计数并产生周期中断的外围设备。用户可以使用定时器器件支持很多与时间有关的设备，如 HAL 系统时钟，报警，时间测量等。通过使用定时器设备，Nios II 处理器必须拥有外部定时器。

HAL API 支持两种类型的定时器器件。

1．系统时钟驱动器。支持报警。

2．时间信息驱动。支持高分辨率的时间测量。

一个定时器既可以用作系统时钟驱动器，也可以用于时间信息驱动，但不可以同时拥有这两种功能。

系统时钟驱动器

HAL 系统时钟驱动器提供周期信号，使得系统时钟每次加 1。软件可以运用系统时钟在特殊时间运行相应的函数，并获得时间信息。用户可以通过在 Nios II IDE 中的系统库属性中的设置，将外围的特殊定时器用作系统时钟驱动器。

HAL支持下列标准UNIX功能：gettimeofday( )，settimeofday( )，和times( )。基于HAL系统时钟，通过使用如上函数返回时间信息。

系统时钟使用ticks组件进行时间测量。对于硬件及软件的嵌入式工程师来说，千万不要将HAL系统时钟和驱动Nios II处理器的时钟信号相混淆。HAL系统时钟普遍长于硬件系统时钟。头文件system.h定义了时钟频率。

运行时，用户可以通过alt_nticks( )函数得到当前的系统时钟值。返回值为系统时钟记录的系统复位开始到当时的时间。可以调用alt_ticks_per_second( )函数，通过每秒系统时钟的变化次数来计算系统时钟的频率。

同样也可以通过UNIX函数gettimeofday( )来获得当前时间。但是事先必须调用settimeofday( )对时间进行校对。另外，用户还可以通过times( )函数的到过去时钟的变化次数。这些函数得源文件在times.h中。

报警

用户可以使用HAL的alarm程序设置在特定时间运行相应的函数。软件程序中可以调用alt_alarm_set( )函数设定报警

alt_alarm_set( )：

```
int alt_alarm_start (alt_alarm* alarm,alt_u32 nticks,alt_u32 (*callback) (void*
                     context),void* context);
```

在nticks发生后调用callback。当调用callback时，将context作为变量传给callback。按照输入的alarm变量，alt_alarm_start( )将结构体指针初始化。用户不需要自己初始化。

Callback函数可以将报警复位。返回值是到下一次调用callback时经历的时间。如果返回值为零，则意味着用户须将报警功能关闭。可以调用alt_alarm_stop( )函数。

Alarm callback函数在中断的时候执行，当使用alarm callback时必须遵守。下面的代码给出了每秒钟对报警信息进行记录的程序段。

历程: 使用周期 Alarm Callback 函数

```
#include <stddef.h>
```

```
#include <stdio.h>
#include “sys/alt_alarm.h”
#include “alt_types.h”
/*
* callback函数
*/
alt_u32 my_alarm_callback (void* context)
{
    /* 这个函数每秒调用一次 */
    return alt_ticks_per_second( );
}
...
/* The alt_alarm must persist for the duration of the alarm. */
static alt_alarm alarm;
...
if (alt_alarm_start (&alarm,alt_ticks_per_second( ),my_alarm_callback,NULL) < 0)
{
    printf (“No system clock available\n”);
}
```

Timestamp驱动

有时用户想使用比系统提供的HAL系统时钟计数器更加准确的方法对时间进行精准的测量。HAL支持timestamp驱动高分辨率的定时功能。Timestamp驱动提供只加计数器，以获得时间信息。HAL只为系统提供一个timestamp驱动。用户可以通过在Nios II IDE中的设置，使用一个外围定时器作为timestamp。

如果启用timestamp驱动，则可以调用alt_timestamp_start( )和alt_timestamp( )函数。调用alt_timestamp_start( )函数启动定时器，然后调用alt_timestamp( )返回timestamp计数器的时间值。再次调用alt_timestamp_start( )将计数器清零，计数器上限为2^{32}-1。可以通过调用alt_timestamp_freq( )函数得到计数器的速率。这个速率代表Nios II处理器运行的频率——一般每秒几百万个周期。在alt_timestamp.h头文件中对timestamp驱动进行定义。

如下代码列出了使用timestamp对时间进行测量。

```
#include <stdio.h>
#include “sys/alt_timestamp.h”
#include “alt_types.h”
int main (void)
{
    alt_u32 time1;
    alt_u32 time2;
    alt_u32 time3;
    if (alt_timestamp_start() < 0)
    {
        printf (“No timestamp device available\n”);
    }
    else
```

```
    {
        time1 = alt_timestamp( );
        func1( ); /*第一个监控函数*/
        time2 = alt_timestamp( );
        func2( ); /*第二个监控函数  */
        time3 = alt_timestamp( );
        printf (“time in func1 = %u ticks\n”,(unsigned int) (time2 – time1));
        printf (“time in func2 = %u ticks\n”,(unsigned int) (time3 – time2));
        printf (“Number of ticks per second = %u\n”,(unsigned int)
                alt_timestamp_freq( ) );
    }
    return 0;
}
```

使用 Flash 器件

HAL 支持 flash 存储器件。Flash 使用专门的编程协议来存储数据。HAL API 支持写数据到 flash 的函数。例如：您可以使用这些函数对基于 flash 的文件系统进行操作。HAL API 同样支持对 flash 器件的读操作，但不是经常使用。因为对于大多数 flash 器件，程序只是将其作为一个简单的存储器件来读，而不需要调用 HAL API 函数。如果读取 flash 器件时需要特殊的协议，如 Altera 的串行配置芯片 EPCS 系列。只能通过 HAL API 进行读取和写操作。

这一段将具体介绍 HAL API 的 flash 器件模式。下面提供两种不同的方法对 flash 器件进行访问。

■ 访问单一 flash 器件：

■ 访问 Fine-Grained flash 器件

访问 flash 器件的 API 函数在 sys/alt_flash.h 中定义。

访问单一 flash 器件

接口函数有alt_flash_open_dev( )，alt_write_flash( )，alt_read_flash( )，和alt_flash_close_dev( )，可以调用alt_flash_open_dev( )打开flash器件，返回flash文件的编号。这个函数使用flash器件的名称作为单一的变量，如在system.h中定义的一样。如果用户得到了文件编号，用户可以使用alt_write_flash( )函数将数据写入flash，函数原形如下：

int alt_write_flash(alt_flash_fd* fd,int offset,const void* src_addr,int length)

fd为文件的指针，调用函数在offset处开始写入数据，scr_addr为地址指针，length为数据长度。

同样，如下为从flash器件中读取数据，原形为：

int alt_read_flash(alt_flash_fd* fd,int offset,void* dest_addr,int length)

变量定义与int alt_write_flash相同。

使用alt_flash_close_dev( )将文件关闭。

下面的代码说明如何使用上面的函数对名为/dev/ext_flash单一flash器件进行读写。

```
#include <stdio.h>
#include <string.h>
```

```
#include “sys/alt_flash.h”
#define BUF_SIZE 1024
int main ()
{
    alt_flash_fd* fd;
    int ret_code;
    char source[BUF_SIZE];
    char dest[BUF_SIZE];
    /* Initialize the source buffer to all 0xAA */
    memset(source, 0xAA, BUF_SIZE);
    fd = alt_flash_open_dev(“/dev/ext_flash”);
    if (fd!=NULL)
    {
        ret_code = alt_write_flash(fd, 0, source, BUF_SIZE);
        if (ret_code==0)
        {
            ret_code = alt_read_flash(fd, 0, dest, BUF_SIZE);
            if (ret_code==0)
            {
                /*
                * Success.
                * At this point, the flash is all 0xAA and we
                * should have read that all back into dest
                */
            }
        }
        alt_flash_close_dev(fd);
    }
    else
    {
        printf(“Can’t open flash device\n”);
    }
    return 0;
}
```

块擦除

一般来讲，flash器件是分块的，alt_write_flash( )在写入数据前，要先将flash内部的数据擦除。这种情况下，不能保存块中已经存在的信息。当用户不从边界开始写的时候，就会引起很多误擦除。如果要保存flash中已经存在的信息，则可以使用fine-grained flash函数。

表4.3总结了用户在使用简单的flash访问时，是怎样将内部的数据擦除的。表4.3给出的8Kb的flash有两个4Kb的块组成。首先从地址为0x0000位址开始写入5Kb的0xAA。然后再从0x1400开始写入2Kb的0xBB。在第一次写入成功后（在time(2)）flash内部有5Kb的0xAA，剩余的空间为0xFF。当第二次写入开始时，在写入第二块开始前，第二块已经被擦除了，在time(3)时，flash中的内容为4Kb的0xAA

和4Kb的0xFF。当第三次写入完成后，即时间time(4)，flash中0x1000地址后的2Kb的0xFF被擦除，写入了0xBB。
表4.3

Table 4–3. Example of Writing Flash & Causing Unexpected Data Corruption

Address	Block	Time t(0)	Time t(1)	Time t(2)	Time t(3)	Time t(4)
		Before First Write	First Write		Second Write	
			After Erasing Block(s)	After Writing Data 1	After Erasing Block(s)	After Writing Data 2
0x0000	1	??	FF	AA	AA	AA
0x0400	1	??	FF	AA	AA	AA
0x0800	1	??	FF	AA	AA	AA
0x0C00	1	??	FF	AA	AA	AA
0x1000	2	??	FF	AA	FF	FF *(1)*
0x1400	2	??	FF	FF	FF	BB
0x1800	2	??	FF	FF	FF	BB
0x1C00	2	??	FF	FF	FF	FF

访问Fine-Grained flash器件
有三个辅助函数，支持最高间隔尺寸的flash写入：
alt_get_flash_info( )，alt_erase_flash_block( )和alt_write_flash_block( )。
就flash的性质而言，用户不可单独擦除一块中单独某一个地址的内容，而必须一次将整块的flash擦除。写入flash只是将每一位的1改为0。如果想要将任何一位的1改为0，则必须将整个块。
所以，如果想要进修改某一位的信息，则需先将flash中整块的信息读到缓冲存储中，然后再缓冲中将其修改，然后擦除flash块，再将缓冲中修改后的信息重新写回flash块中。Fine-Grained flash访问函数自动的执行这些相关操作。
alt_get_flash_info( )获得要擦除的区域，每个区域擦除的块数，和擦除块的大小，函数原形：
int alt_get_flash_info(alt_flash_fd* fd,flash_region** info,int* number_of_regions)
调用完成后，由number_of_regions返回的地址中，包含有擦除的区域数量，*info指针指向flash_region结构体。Flash_region结构体在sys/alt_flash_types.h中有定义。

```
typedef struct flash_region
{
    int offset;                 /*  地址初始的偏移量  */
    int region_size;            /*  擦除区域的大小  */
    int number_of_blocks;       /*  区域的块数  */
    int block_size;             /*  区域中每一个块的大小  */
}flash_region;
```

alt_erase_flash_block( )单独的块擦除，其原形为：
int alt_erase_flash_block(alt_flash_fd* fd,int offset,int length)
fd指向要擦除的flash存储器，每一块有offset个字节组成，块长为length。
alt_write_flash_block( )单独的写入函数，其原形为：
int alt_write_flash_block(alt_flash_fd* fd,int block_offset,int data_offset,const void

*data,int length)
fd指向要写入的flash，写入block_offset个字节，在data指向的位置开始写length的字节数。

下面的例子给出了如何使用fine-Grained flash访问：

```
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 100
int main (void)
{
    flash_region* regions;
    alt_flash_fd* fd;
    int number_of_regions;
    int ret_code;
    char write_data[BUF_SIZE];
/* Set write_data to all 0xa */
    memset(write_data, 0xA, BUF_SIZE);
    fd = alt_flash_open_dev(EXT_FLASH_NAME);
    if (fd)
    {
        ret_code = alt_get_flash_info(fd,
        &regions,
        &number_of_regions);
        if (number_of_regions && (regions->offset == 0))
        {
            /* Erase the first block */
            ret_code = alt_erase_flash_block(fd,
            regions->offset,
            regions->block_size);
            if (ret_code)
            {
                /*
                * Write BUF_SIZE bytes from write_data 100 bytes into
                * the first block of the flash
                */
                ret_code = alt_write_flash_block (
                fd,
                regions->offset,
                regions->offset+0x100,
                write_data,
                BUF_SIZE );
            }
        }
    }
```

```
    return 0;
}
```

使用 DMA 器件

HAL 支持直接存储器访问器件（DMA），一般用于大量数据传输，数据源和目标器件可以是存储器，也可以是存储器件也可以是其它器件，如以太网连接等等。在 HAL 的 DMA 模式中，DMA 传输分为两类：发送和接收。因此，HAL 支持两种器件驱动支持发送和接收通道。发送是将数据从主机缓冲区将数据发送到目标器件，接收器件将在主机接收的数据存储到接收器缓冲区。由于依靠底层硬件实现这一功能，所以上层软件只需访问两个端点的其中一个。
如图 4.2 所示，给出了基本的 DMA 传输示意图。从一个存储器将数据复制到另一个存储器将同时用到 DMA 发送和接收两个功能。

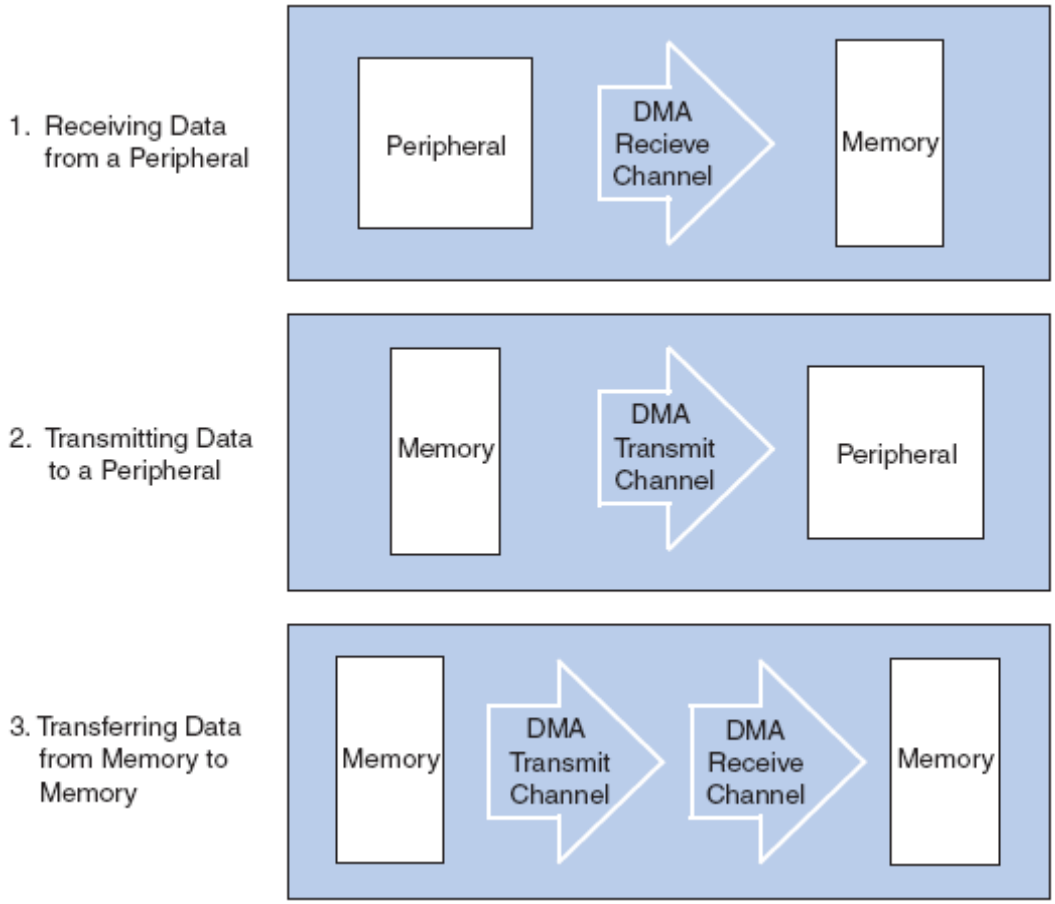

图 4.2 DMA 传输基本类型

相对于 DMA 器件的 API 在 sys/alt_dma.h 中定义。由于 DMA 操作的时物理存储，所以需要考虑到相互的高速缓存。

DMA 发送通道

按 DMA 发送器件编号将 DMA 的发送请求编入队列。实现这一操作需要 alt_dma_txchan_open()函数。这个函数只是用在 system.h 中定义的器件名称这一个变量。

下面这段代码介绍了如何为 DMA 发送期间 dma_0 获得这一编号。

```
#include <stddef.h>
#include “sys/alt_dma.h”
int main (void)
{
    alt_dma_txchan tx;
    tx = alt_dma_txchan_open (“/dev/dma_0”);
    if (tx == NULL)
    {
        /* Error */
    }
    else
    {
        /* Success */
    }
    return 0;
}
```

alt_dma_txchan_send()函数将利用这一编号记入一个发送请求。函数原形如下：

```
typedef void (alt_txchan_done)(void* handle);
int alt_dma_txchan_send (alt_dma_txchan dma,const void* from,alt_u32
                                length,alt_txchan_done* done,void* handle);
```

调用alt_dma_txchan_send()函数为DMA通道记入发送请求。Length变量确定发送的数据位数，from变量确定发送数据的源地址。函数在DMA完全处理完成之前返回值。返回值显示是否已经成功进入队列。如果返回值为负数则说明请求失败，当传输完成后，调用用户支持函数done，handle变量显示作为标志信息。

另外系统提供两个用于DMA传输的函数，alt_dma_txchan_space()和alt_dma_txchan_ioctl()。alt_dma_txchan_space()函数返回传输队列中附加的发送请求的数量。alt_dma_txchan_ioctl()函数为发送器件执行特殊器件操作。

DMA接收通道

DMA接收通道的操作与发送通道相似。通过使用alt_dma_rxchan_open()函数为DMA接收通道获得编号。然后使用alt_dma_rxchan_prepare()函数发送接收请求。alt_dma_rxchan_prepare()函数原形如下：

```
typedef void (alt_rxchan_done)(void* handle, void* data);
int alt_dma_rxchan_prepare (alt_dma_rxchan dma,void* data,alt_u32
                                length,alt_rxchan_done* done,void* handle);
```

调用这一函数为DMA通道发送请求。有length长度的数据需要放置在data地址上，函数在DMA传输完成前返回值。通过返回值判断请求是否成功被排入队列。返回如果为负值，则说明请求失败。当传输完成，用户支持函数done，handle变量提供标志信息和一个指向接收数据的指针。

同样接收通道提供两个附加函数alt_dma_rxchan_depth()，alt_dma_rxchan_ioctl()。alt_dma_rxchan_depth()返回可以排入队列的最大值。alt_dma_rxchan_ioctl()函数为接收通道执行特殊器件操作。

下面的例程提供一个完整的发送DMA接收请求的实例。

```
#include <stdio.h>
```

```
#include <stddef.h>
#include <stdlib.h>
#include “sys/alt_dma.h”
#include “alt_types.h”
/*  标志位用于显示传输已经完成  */
volatile int dma_complete = 0;
/*  传输完成时调用函数  */
void dma_done (void* handle, void* data)
{
    dma_complete = 1;
}
int main (void)
{
    alt_u8 buffer[1024];
    alt_dma_rxchan rx;
    /*  为器件获得编号*/
    if ((rx = alt_dma_rxchan_open (“/dev/dma_0”)) == NULL)
    {
        printf (“Error: failed to open device\n”);
        exit (1);
    }
    else
    {
        /*  发送接收请求*/
        if (alt_dma_rxchan_prepare (rx, buffer, 1024, dma_done, NULL)< 0)
        {
            printf (“Error: failed to post receive request\n”);
            exit (1);
        }
        /*  等待传输完成*/
        while (!dma_complete);
        printf (“Transaction complete\n”);
        alt_dma_rxchan_close (rx);
    }
    return 0;
}
```

存储器到存储器的DMA传输

从一个存储器的缓冲区将数据复制到另一个存储器的缓冲区需要用到DMA接收和发送驱动。下面的代码显示了将接收请求编入传输请求队列来完成从存储器到存储器的DMA传输的过程。

```
#include <stdio.h>
#include <stdlib.h>
#include "sys/alt_dma.h"
#include "system.h"
```

```
static volatile int rx_done = 0;
/*
* Callback function that obtains notification that the data has
* been received.
*/
static void done (void* handle, void* data)
{
      rx_done++;
}
/*
*
*/
int main (int argc, char* argv[], char* envp[])
{
      int rc;
      alt_dma_txchan txchan;
      alt_dma_rxchan rxchan;
      void* tx_data = (void*) 0x901000; /* pointer to data to send */
      void* rx_buffer = (void*) 0x902000; /* pointer to rx buffer */
      /* Create the transmit channel */
      if ((txchan = alt_dma_txchan_open("/dev/dma_0")) == NULL)
      {
            printf ("Failed to open transmit channel\n");
            exit (1);
      }
      /* Create the receive channel */
      if ((rxchan = alt_dma_rxchan_open("/dev/dma_0")) == NULL)
      {
            printf ("Failed to open receive channel\n");
            exit (1);
      }
      /* Post the transmit request */
      if ((rc = alt_dma_txchan_send (txchan,tx_data,128,NULL,NULL)) < 0)
      {
            printf ("Failed to post transmit request, reason = %i\n", rc);
            exit (1);
      }
      /* Post the receive request */
      if ((rc = alt_dma_rxchan_prepare (rxchan,rx_buffer,128,done,NULL)) < 0)
      {
            printf ("Failed to post read request, reason = %i\n", rc);
            exit (1);
      }
      /* wait for transfer to complete */
```

```
    while (!rx_done);
    printf ("Transfer successful!\n");
    return 0;
}
```

缩减代码

代码量的大小通常是开发者比较关心的一个问题，因为这涉及到消耗存储器存储量的大小。要想控制这个消耗，控制和缩减代码量非常重要。
HAL 环境提供给用户可以控制和缩减代码量，如果 Nios II 硬件系统精确的包含用户程序控制的外围设备。HAL 包含必要的硬件驱动，没有多余的部分。
下面这节介绍了要将代码精简到最简所需要考虑的项目。
使用精简器件驱动
一些器件提供两种驱动代码，一个完整形“fast”和一个简化型“small”。默认模式下HAL系统库使用fast型。用户可以通过在Nios II IDE中选中Use Small Footprint Drivers选项使代码精简。在建立HAL系统库时，用户可以使用–DALT_USE_SMALL_DRIVERS预处理操作。
如表4.4列出了在Nios II系统外围设备中普遍支持Small Footprint驱动代码。Small Footprint选项同样可能影响到系统其它外围设备。

Table 4–4. Altera Peripherals Offering Small Footprint Drivers

Peripheral	Small Footprint Behavior
UART	Polled operation, rather than IRQ-driven.
JTAG UART	Polled operation, rather than IRQ-driven.
Common flash interface controller	Driver is excluded in small footprint mode.
LCD module controller	Driver is excluded in small footprint mode
EPCS serial configuration device	Driver is excluded in small footprint mode

简化文件描述信息库
访问字符模式器件和文件的文件描述信息由文件描述库分配。通过在 Nios II IDE 中 Max file descripter 的属性中设置，可以控制文件描述信息库的大小。同样也可以使用 GNU 指令，使用编译时间常量 ALT_MAX_FD。默认值为 32。
使用/dev/null
在启动时，标准输入，标准输出和标准误差直接指向空器件。由于这种指向，使得在驱动初始化时调用 printf()对系统没有影响。一旦所有的驱动都安装完成，上面的标准输入，输出，误差又重新指向由 HAL 配置过的各个通道。而执行这种重新指向的代码量很少，而且用户可以通过选择 stdin，stdout，stderr 而将执行重新指向的代码删除。如果选择删除哪部分代码，则意味着放弃使用标准输出和标准误差进行数据传输，并且从来不是用 stdin 输入数据。用户可以在 Nios II IDE 中将 stdin，stdout，stderr 作为系统库属性来控制。
使用小规模的文件 I/O 库
使用简版 newlib C 语言库。
在嵌入式系统中很多情况下不需要完整的 newlib C 语言库，GCC 支持缩减 newlib ANSI C 语言库，减少那些嵌入式系统中一般用不到的多余部分。简化的 newlib

工具的代码量更小。用户可以将 newlib 工具作为 Nios II IDE 的系统库选项来控制。当使用 nios2-elf-gcc 指令模式时，-msmallc 指令可以使能简版的 C 语言库。如表 4.5 总结了 Nios II 简版 C 语言库的限制。

表 4.5 Small C 语言库的限制

限制	受影响的函数
printf()函数不支持浮点型。列出的函数不支持%f 和%g（这些函数为 Nios II 的扩展函数，GCC 在 Small C 语言库中不支持这些函数）	asprintf() fiprintf() fprintf() iprintf() printf() siprintf() snprintf() sprintf()
vprintf()程序不支持浮点型数据。列出的函数不支持%f 和%g	vasprintf() vfiprintf() vfprintf() vprintf() vsnprintf() vsprintf()
不支持 scanf()程序. 右侧列出的不支持。	fscanf() scanf() sscanf() vfscanf() vscanf() vsscanf()
不支持查找	fseek() ftell()
不支持打开或关闭文件。只有先前已经打开的 stdout, stderr 和 stdin 可用。	fopen() fclose() fdopen() fcloseall() fileno()
stdio.h 输出程序没有缓冲。	functions supported with no buffering: fiprintf() fputc()

	fputs() perror() putc() putchar() puts() printf() functions not supported: setbuf() setvbuf()
没有 stdio.h 输入程序	fgetc() gets() fscanf() getc() getchar() gets() getw() scanf()
不支持现场	setlocale() localeconv()
因为上面的函数都不支持，所以不支持 C++。	

引导程序序列和入口

正常情况下程序从 main( )函数开始执行。但在 Nios II 中可以选择从 alt_main( )开始执行，从而可以更好的控制启动引导程序。从 main( )和 alt_main( )启动的不同之处在于主机与不同标准应用的区别。

主机与自由标准应用

ANSI 标准 C 中定义了主机标准应用从 main( )开始执行。在 main( )开始的时候，主机应用程序推测执行时刻的环境和所有系统服务已经初始化完成。在 HAL 环境中是这样执行的。但如果用户现在使用 Nios II 编程设计，HAL 的主机运行环境帮助用户加速这一环节的运行。用户不需要考虑系统中包含的硬件，以及如何对其初始化等等，由 HAL 将初始化整个系统。

ANSI 标准 C 同样提供一种可以避免自动初始化的入口，假设用户需要手动的初始化需要使用的器件。Alt_main( )函数提供一个自由标准环境，使用户可以完全的控制系统的各个部分的初始化。自由标准化运行环境减轻了系统的负担，可以使用户手动的初始化需要使用的器件。例如：在 alt_main( )没有将字节型器件初始化，并把 stdout 指向该器件之前调用 printf( )是非法的。

基于 HAL 的启动序列

HAL 为系统提供初始化代码帮助用户执行如下启动序列操作：

- 刷新指令和数据高速缓存区
- 配置堆栈指针
- 配置全局指针寄存器
- 使用提供的连接器标志_bss_start和_bss_end，将BSS区域初始化为零。这两个标志分别为指向BSS区域开始和结尾的指针。
- 如果系统当前没有装载程序，将运行地址在RAM中连接程序段拷贝到RAM中，如rwdata，rodata
- 调用alt_main( )

默认情况下，HAL为alt_main( )提供执行如下操作的函数。

- 调用ATL_OS_INIT( )执行系统具体的初始化功能。如果系统中没有OS调度程序，则这个宏指令不起作用。
- 在操作系统中使用HAL，需要初始化alt_fd_list_lock信息标志，因为他控制着对HAL的访问。
- 初始化中断控制器，并打开中断。
- 调用alt_sys_init( )函数，初始化系统所有硬件及软件模块。Nios II IDE为每一个HAL系统库创建并管理alt_sys_init.c文件。
- 重新指向标准I/O口，是用适当的器件。
- 是用_do_ctors( )函数，调用C++构造程序。
- 在系统关闭时调用C++解除程序。
- 调用main( )
- 调用exit( )，把main( )函数的返回值作为变量传给exit( )。

Nios II EDS安装后产生alt_main.c文件，

用户制定启动序列

在Nios II IDE工程中用户可以自己定义alt_main( )中的启动序列功能。这样使得用户可以完全控制启动程序，可以有选择的使能HAL功能。如果您的应用程序需要alt_main( )接入点，则可以将默认的功能拷贝，并作以修改为系统需要的功能。alt_main( )调用 main( )函数，在 main( )函数返回后，alt_main( )默认进入死循环。调用 exit( )结束死循环。不需要 return 语句。

alt_main( )的函数原形如下：

void alt_main(void)

HAL 编译环境的特点是所有的源代码和文件放在一个搜索路径下。编译系统时首先搜索系统库工程的路径。这使得用户忽略了系统为用户应用工具提供的默认的器件驱动和系统代码。例如，将用户自定义的 alt_sys_init.c 放置到了系统工程路径下，用于支持用户的程序。用户自定义文件优先权高于自动生成文件。

存储器的使用

这一章主要介绍了在物理存储器上，HAL 如何使用存储器，以及 HAL 如何组织代码，数据，堆栈和其他逻辑存储单元。

存储段

默认情况下，基于 HAL 的系统由 Nios II IDE 创建并管理的自动生成的脚本。这些连接脚本控制着在存储器映射的代码和数据。自动生成的连接脚本创建标准的代码和数据段，为系统中每一个物理存储器件加一段。例如，如果在 system.h

中有名为 sdram 的器件，那么就有一个名为.sdram 的存储段。如图 4.3 所示的典型的 HAL 连接图组织。

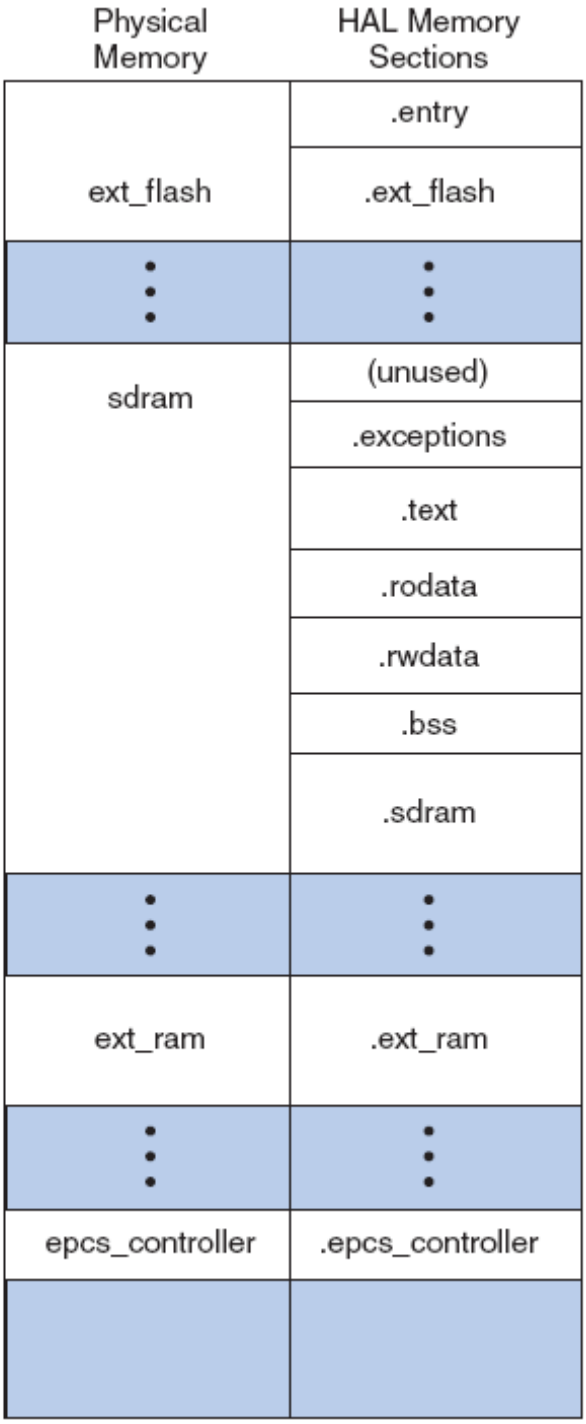

图 4.3 HAL 连接图

包含有Nios II处理器复位和特殊地址的存储器是一种特殊情况，Nios II工具在复位开始的代码段中建立32byte的.entry区域，这个区域专门用来保存复位代码。同样，在特殊地址中建立.exceptions区域。

在包含复位和特殊地址的存储器中，在.entry和.exceptions两个区域之上不再建立保存区域。在.entry或者.exceptions之下的区域，对于Nios II 程序来说不可用。图4.3给出了在.exceptions之下的不可用区域。

位存储器划分代码段和数据段

这一段主要介绍了怎样在特殊存储器区域放置代码段和数据段。一般来说，Nios II IDE自动为其分配合适的区域。但是在特殊情况下，用户可能会想改变其分配。例如要增强性能时，一般将对性能起关键作用的代码放到RAM中，这样可以得到更快的访问速度；当用户想要从RAM中直接复位处理器，但是软件却从flash存储器中启动，等等。这时候就需要手动的将代码放置到合适的位置。

1）初级的设置操作

复位代码一般放置在.reset部分的底部。包含特殊地址的部分一般将特殊的操作代码放置在前面。默认情况下，一般将剩余的代码和数据放置在以下部分：

- .text—所有剩下的代码
- .rodata—只读数据
- .rwdata—可读写的数据
- .bss—初始化为零的数据

用户可以在Nios II IDE的System Library Properties选项中设置.text，.rodata，.rwdata或其他存储器的位置。

2）高级的设置操作

用户可以将自己的每一段代码放到指定的存储区中。在C或C++中，用户可以使用section属性。下面的代码举例说明了如何将foo变量和bar( )函数分别放到on_chip_memory和名为sdram的存储器中。

例程：

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".on_chip_memory.rwdata"))) = 0;
void bar __attribute__ ((section (".sdram.txt"))) (void);
void bar (void)
{
    foo++;
}
```

假设您使用.section指令，例如，下面的代码将所有的代码放到了on_chip_memory中。

```
.section .on_chip_memory.txt
```

堆和栈的放置

默认模式下，堆和栈放在同一个如.rwdata的存储区中。在.rwdata存储区中，栈是从高地址向低地址，堆是从低地址到高地址。在Nios II IDE中，可以将其作为系统库属性进行设置。

默认模式下，HAL不能执行堆和栈的搜索。这可以使函数调用和存储器分配更快，但这就意味着无法使用malloc( )（在C中）和new( )（在C++中）对堆进行检查。用户可以在Nios II IDE中的System Library Properties中设置打开run-time栈检测。当栈检测打开时，malloc( )和new( )可以对堆进行检测。

设置堆容量大小时，采用十进制将ALT_MAX_HEAP_BYTES设置为最大。如：-DALT_MAX_HEAP_SIZE=1048576将堆的大小设置为0x100000。也可以在Nios II IDE中设置这一选项。

堆栈检测占用系统资源。如果将堆栈检测功能关闭，那么在使用代码时必须使其能在限制的堆栈大小下完成其相应的操作。

全局指针寄存器

在Nios II程序中，全局指针寄存器可以使快速访问全局数据结构。Nios II编译器自动使用全局指针访问其指定的数据结构，不需要用户做任何操作，除非用户改变默认的编译器选项。
全局指针可以访问连续的64Kb的空间。为了防止这个地区产生溢出，使用小的全局数据结构，所谓的“小”的数据结构，只的是小于8bytes。
小的数据结构体分配在小的全局数据段中.sdata，.sdata2，.sbss，.sbss2。小的全局数据段被分为.rwdata和.bss部分。他们相互联系在一起，如图4.4所示，允许全局指针访问。

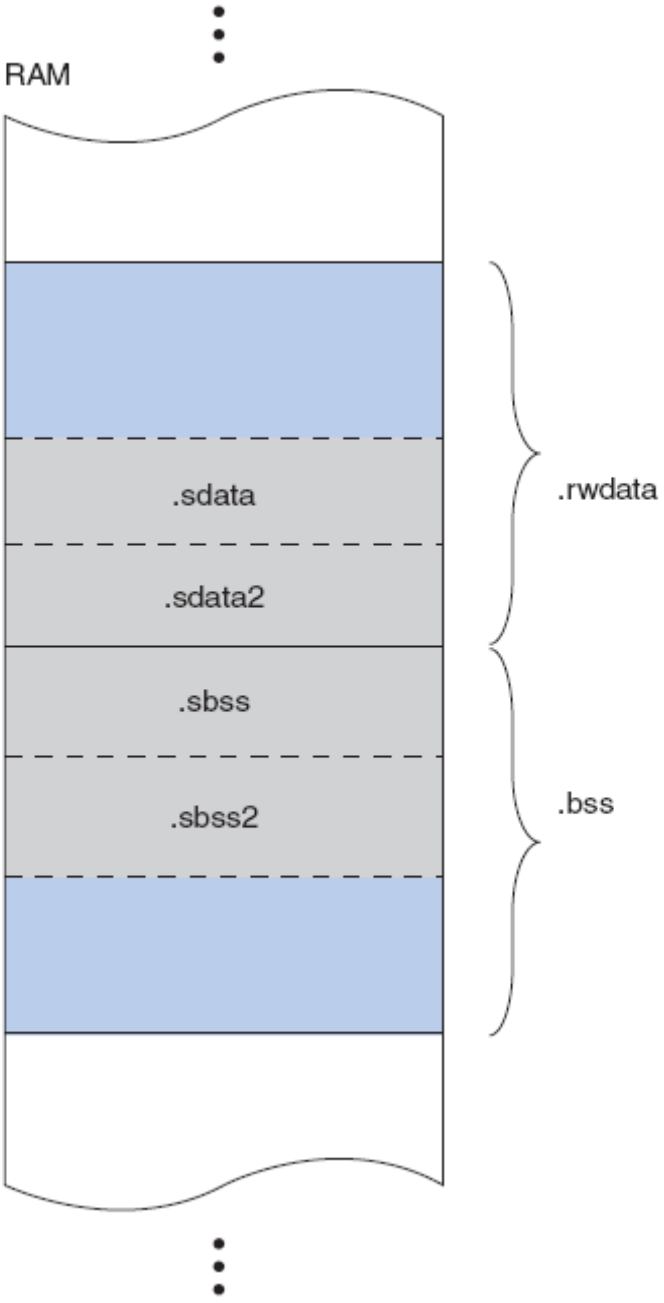

图4.4
如果所有小型的全局数据结构体的总合超过了64Kbytes，超出了全局指针的范围，则连接器发出错误信息“Unable to reach <variable name> ...from the global pointer ... because the offset ... isout of the allowed range, -32678 to 32767。”

用户可以使用-G操作来确定全局数据结构体的大小，如，-G 4限定全局指针使用4bytes空间的全局数据结构体，这个操作变量采用的是十进制。

启动模式

处理器的启动存储器指的是包含复位向量的存储器。这个器件可以是外部flash，Altera的EPCS串行配置芯片，或者是片上RAM。不论启动存储器的特性如何，系统是以HAL为基础，所以程序和数据段开始都存在这个存储器中。HAL支持较小的启动程序，在启动时将程序段拷贝到运行单元。用户可以在系统库属性页面上为程序和数据存储器指定运行单元。

如果.text运行时间单元在启动存储器之外，则Altera的Nios II IDE flash编辑器会在复位地址上装载启动装载程序，负责在调用_start前装载所有的程序和数据。当从EPCS中启动时，有硬件完成这一装载程序。

但是如果.text运行时间单元在启动存储器中，则系统不需要分离的启动装载程序，HAL可以直接使用_reset入口。_reset函数初始化指令缓冲区，并调用_start函数。这个初始化序列可以使用户的程序直接在flash存储器中启动或者运行。

当运行这种模式时，HAL负责将所有启动的程序装载到RAM中。.rwdata，.rodata和.exceptions存储段在调用alt_main( )函数前全部被装载，调用alt_load( )函数实现这一装载，如果需要装载另外的部分，则调用alt_load_section( )函数。

HAL 系统库文件路径

用户可能希望看到 HAL 系统库中的文件，特别是头文件，用于参考。但不要编辑 HAL 系统库文件。

HAL 文件的位置

HAL 系统库文件在若干个目录下，这是 Nios II 系统的风格决定的。每一个 Nios II 系统含有很多个不同的外围设备，因此对于每一个系统的 HAL 系统库都不同，您可以在下面的位置找到相关的 HAL 文件：

■ <Nios II EDS 安装路径>/components 包含大多数HAL系统库文件

■ <Nios II EDS 安装路径>/components/altera_hal/HAL/inc/syscontains定义的HAL通用器件模式的头文件。使用#include指令，涉及这些文件的在 <Nios II EDS 安装路径>/components/altera_hal/HAL/inc/. 例如，要包含DMA器件，则使用#include sys/alt_dma.h

■ 每一个Nios II IDE系统工程目录下包含system.h文件，为系统库工程使用。

■ <Nios II EDS 安装路径>/bin 包含newlib ANSI C库头文件。

■ <Quartus® II Root Directory>/sopc_builder/components为SOPC Builder组件提供HAL驱动。

参考资料

Nios II Softwear Developer’s Handbook Altera Corporation May 2006