xTIMEcomposer 用户指南

版本 B

出版日期：2013 年 11 月 14 日

目录

第 A 部分
安装

内容

xTIMEcomposer 运行系统要求

支持 xTIMEcomposer 工具的官方平台：

Windows XP SP3

·具 32 位 JAVA 运行环境的 32 位系统

Windows 7 SP 1

·具 32 位 JAVA 运行环境的 32 位系统

·具 32 位 JAVA 运行环境的 64 位系统

Mac OS X 10.5 +

· 英特尔处理器

Linux CentOS 5.8

· 具 32 位 JAVA 运行环境的 32 位系统

· 具 64 位 JAVA 运行环境的 64 位系统

xTIMEcomposer 工具还可用于其他多种版本的 Linux 系统中，如 RedHat 和 Ubuntu。关于兼容版本的最新信息，请看：

· http://www.xmos.com/tools

你必须安装 1.5 或以上的 JAVA 运行环境版本，可以从下面网址下载：

· http://java.sun.com/javase/downloads

2 安装说明

本章内容：
·工具安装
· USB 驱动安装

xTIMEcomposer 及相关驱动为特定平台的可下载文件形式。

2.1 工具安装

按下列步骤在个人电脑上安装工具

Windows 系统：

1. 从下列地址下载 Windows 安装程序
   · http://www.xmos.com/tools
2. 双击运行安装程序，按照屏幕提示在个人电脑上安装工具。

Mac 系统：

1. 从下列地址下载 Macintosh 安装程序
   · http://www.xmos.com/tools
2. 双击打开下载的 Installer，将 xTIMEcomposer 图标拖到系统应用文件夹下。
   安装程序将文件拷贝至你硬盘中。
3. 卸载安装程序

Linux 系统：

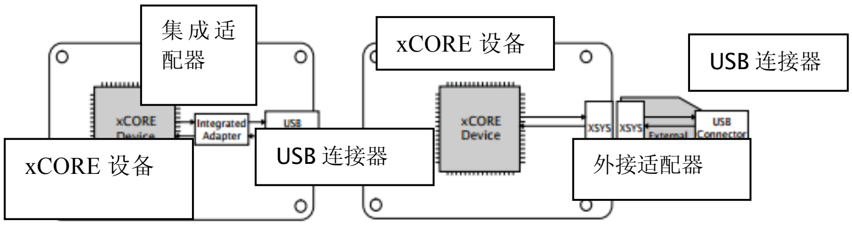
1. 从下列地址下载 Linux 档案库
   · http://www.xmos.com/tools
2. 将档案库解压到安装目录下，例如输入下列命令：

```
▶ tar -xzf archive.tgz -C /home/user
```

2.2 USB 驱动安装

xTIMEcomposer 与 USB 上方的开发板相接。一些开发板带有集成调试适配器，而另一些则需要外接适配器通过 XSYS 链接器与开发板链接，如图 1 所示。

[redacted]
开发板的适配器控
[redacted]



xTIMEcomposer 支持基于 FTDI 或 xCORE USB 至 JTAG 芯片的适配器。请参照开发板操作手册确定使用的驱动器。

Windows 系统：
使用工具安装程序安装 JTAG 驱动。安装后插入 xCORE 开发板以加载驱动。

Mac 系统：
OS X 本身就提供 USB 驱动支持。

Linux 系统：
一些版本的 Linux 本身就提供 USB 驱动支持，而于另一些版本中则必须启用驱动（见 XM-001514-PC）。

第 B 部分
快速启动

内容

3　启动 xTIMEcomposer

本章内容

· 启动 xTIMEcomposer Studio
· 启动命令行工具
· 快速启动菜单

3.1 启动 xTIMEcomposer Studio

启动 xTIMEcomposer Studio：

Windows 系统：

选择开始——程序——XMOS——xTIMEcomposer_12——xtimecomposer

OS X 系统：

打开一个新的查找窗口，导航到应用文件夹，打开文件夹 XMOS_xTIMEcomposer_12，双击 xtimecomposer.app 图标。

Linux 系统：

打开终端窗口，变更至安装目录并输入下列命令：

· source　SetEnv
· xtimecomposer

3.1.1 xTIMEcomposer 注册

首次启动 xTIMEcomposer Studio 时，你需要采用 XMOS 账户进行工具注册，因此你需要联网。

注册 xTIMEcomposer 后，你可以离线使用工具，尽管我们建议你尽可能保持联网以下载最新版本的软件和文档。

注册好处包括文档自动通知、软件直接于 Studio 中更新，此外还可以选择从工具中直接管理账户设定。

图 2：xTIMEcomposer Studio 注册界面

3.2 启动命令行工具

xTIMEcomposer 命令行工具使用一整套环境变量（见§ 9.8）来搜索标头文件、库及目的设备。在路径中添加 xTIMEcomposer 工具，并配置默认环境变量。

Windows 系统：

选择“开始”——“程序”——XMOS——xTIMEcomposer_12——“命令提示符”

OS X 系统：

打开终端窗口，变更至安装目录，或从 Finder 窗口打开并输入下列命令：

· SetEnv.command

Linux 系统：

打开终端窗口，变更至安装目录并输入下列命令：source SetEnv

输入工具名称和命令行选项，可以运行任意一种工具。下文中概述了一些常用命令。

3.3 欢迎界面

xTIMEcomposer Studio 中的欢迎界面为所有用户，包括 XMOS 新开发人员和经验丰富的用户，提供便捷的起始点。

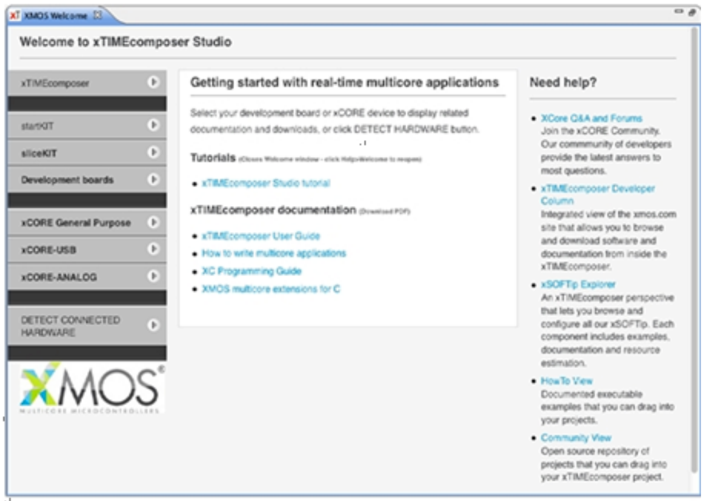


图 3 xTIMEcomposer Studio 欢迎界面

3.4 开发者专栏

xTIMEcomposer 开发者专栏包含一个 web 浏览器，显示开发者编写实时多核程序时所需要的信息，包括：

XMOS 网站的即时显示（xmos.com），允许用户从开发工具中下载最新软件和文档，并管理用户自己的 XMOS 账户；

自 xTIMEcomposer 视图所选择的 xCORE 资源相关的文档和示例，如 xSOFTip 或 HowTo

示例；

显示如何使用 XMOS 开发工具和实时多核心应用程序的交互式教程同样显示在开发者专栏中。通过帮助——教程将教程加载至开发者专栏中。

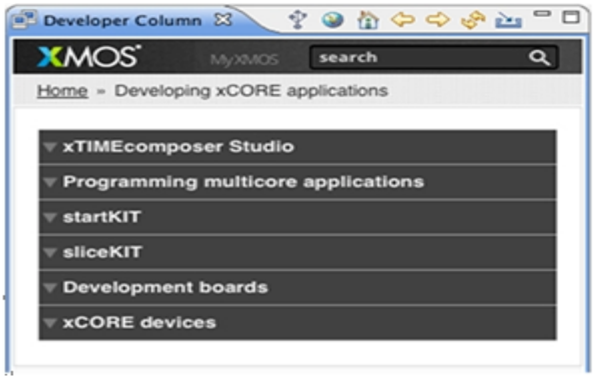


图 4 xTIMEcomposer Studio 开发者专栏

开发者通过开发者专栏工具条来进行内容导航，提供下列选项：

监测：校验连接硬件，显示与硬件相关的最新版本的软件和文档；无可用硬件时，将显示最新的 xTIMEcomposer 软件版本。

主页：载入开发者专栏的默认主页，显示与工具、开发板和硅器件相关的内容。

本地主页：为 xTIMEcomposer 加载主页以及采用离线更新选项所离线下载和储存的编程内容。

后退和前进：开发者专栏中下一页和上一页之间的导航。

刷新：重新加载当前网页

更新下载至本地驱动器的 xTIMEcomposer 和 C 程序的离线数据，从而离线使用 xTIMEcomposer 开发者专栏中的数据。

通过窗口——显示视图——开发者专栏，可随时显示开发者专栏。

4　常用命令

本章内容：

· XCC

· XRUN
· XGDB
· XSIM·

本文概述了一些常用命令，可通过命令行运行。

4.1 XCC

为开发板编译程序时，输入下列命令：

1. xcc -print-targets

XCC 显示了一系列支持的开发板。

2. xcc <file> -target=<board> -o <binary>

XCC 编译文件时为目的板生成二进制可执行文件。

4.2 XRUN

输入下列命令，在开发板上加载编译程序。

1. xrun -l

XRUN 打印出一份与个人电脑链接的所有 JTAG 适配器以及 JTAG 链上所有设备的枚举列表，表格如下：

标识	名称	适配器标识	设备
--	----	----	----

2. xrun --id <n> --io <binary>

XRUN 将二进制文件加载于与带规定标识符的适配器链接的硬件上。

假定从硬件到终端的为标准输出流，则--io 选项使 XRUN 可以与适配器保持链接。

4.3 XGDB

输入下列命令，编译并调试程序：

1. xcc <file> -target=<board> -o <binary> -g

启用调试信息后，XCC 会对文件进行编译。

2. xgdb bin.xe

GDB 根据提示加载。

3. list-devices

GDB 打印出一份与个人电脑链接的所有 JTAG 适配器以及每根 JTAG 链上所有设备的枚举列表，表格如下：

标识	名称	适配器标识	设备
--	----	----	----

4. connect --id <id>

GDB 与目的硬件链接。

5. 加载

GDB 加载二进制文件。

6. break main

GDB 在主要函数中添加一个转效点。

7. continue

GDB 运行程序直到达到主要函数。

4.4 XSIM

输入下列命令在模拟器上运行程序：

· xsim <binary>

根据 GDB 提示从调试器启动模拟器：

· connect -s

你可以按照使用开发板时的方式在模拟器上加载程序。

第 C 部分
在 XDE 中开发

内容

5 在 xTIMEcomposer 中导入或导出项目

本章内容

· 导入 xSOFTip 组件

· 导入 HowTo 示例

· 导入社群项目

· 导入 xTIMEcomposer 项目

· 导出 xTIMEcomposer 项目

xTIMEcomposer Studio 具有一组视图，你可以采用视图将软件自不同软件源导入你的项目。

xSOFTip：显示 xSOFTip 组件及功能区块的 XMOS 库，配置组件并将组件拖拽至你的项目中。

HowTo：显示示例和代码片断库，显示如何于多核应用程序中完成常规任务，然后将代码拖拽至你的项目中。

社群：于 GitHub 开放资源中显示项目，并将其拖拽至你的项目中。

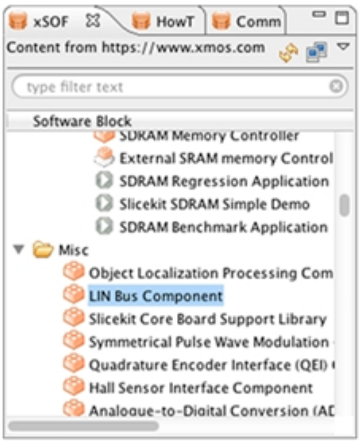


图 5 xSOFTIP 视图

所有 xSOFTip 组件、HowTo 示例和社群项目都包含相关文档和代码使用示例。

你可以通过 xTIMEcomposer 导入/导出功能与其他开发人员共享 xTIMEcomposer 项目。

5.1 导入 xSOFTip 组件

按照下列步骤导入 xSOFTip 组件：

1. 在 xTIMEcomposer 窗口左下角选择 xSOFTip 视图（窗口——显示视图——xSOFTip）
2. 双击想要导入的组件
3. 选择所需版本；默认选择最新版本。
4. 点击“完成”

   开发者专栏中显示了关于 xSOFTip 组件的文档。

5.2 导入 HowTo 示例

按照下列步骤导入 xSOFTip 组件：

1. 在 xTIMEcomposer 窗口左下角选择 HowTo 视图（窗口——显示视图——HowTo）
2. 双击想要导入的示例
3. 选择所需版本；默认选择最新版本。
4. 点击“完成”

   开发者专栏中显示了关于 HowTo 示例的文档。

5.3 导入社群项目

按照下列步骤导入社群：

1. 在 xTIMEcomposer 窗口左下角选择社群视图（窗口——显示视图——HowTo）
2. 双击想要导入的项目
3. 点击“完成”

   开发者专栏中显示了关于社群项目的文档。

5.4 导入 xTIMEcomposer 项目

1. 选择“文件”——“导入”
2. 双击“常规”选项，选择“现有项目到工作区”，点击“下一步”。
3. 在“导入”对话框中，点击“浏览”（位于“选择归档文件”文本框旁边）
4. 选择归档导入文件，点击“打开”
5. 点击“完成”

5.5 导出 xTIMEcomposer 项目

按照下列步骤导出项目：

1. 选择“文件”——“导出”

2. 双击“常规”选项，选择“归档文件”，而后点击“下一步”。

3. 在左上方面板中选择想要导出的项目，你可以在右上方面板中以反选的形式排除文件。

4. 在“至归档文件”文本框中输入存档文件名。

5. 点击“完成”。

6 使用 xSOFTip 开发应用程序

本章内容：

· xSOFTip 浏览器视图

· 系统信息

· 配置 xSOFTip 组件

· 从应用程序中生成项目

XMOS 提供了含各种界面如 USB、Ethernet 和串行端口和 DSP、协议功能的 xSOFTip 模块选择。xSOFTip 模块使用 xCORE 资源执行规定功能。

为了尽可能简单的选择和配置 xSOFTip，可以使用 xSOFTip 浏览器从 xSOFTip 库中浏览可用模块，了解资源用途并根据自身规格配置模块。工具还可以评估是否选择了适合自身设计的设备。

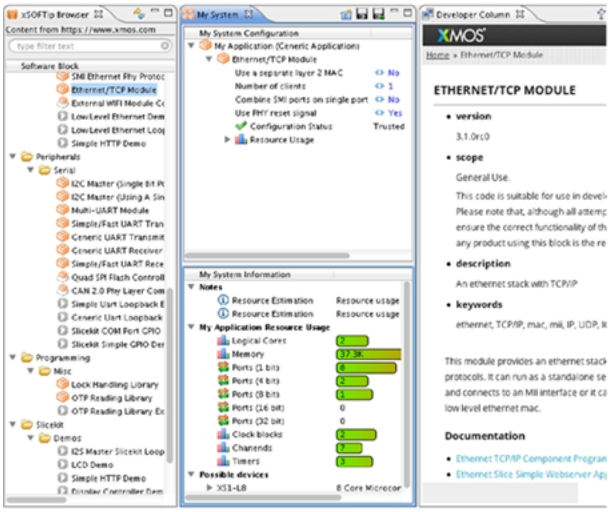


图 6 xSOFTIP 浏览器视图

6.1 xSOFTip 浏览器视图

xTIMEcomposer Studio 独立视图中含 xSOFTip 浏览器。

1. 选择“窗口”——“打开视图”——“XMOS xSOFTip 浏览器”，从而打开 xSOFTip 浏览器视图。

xSOFTip 浏览器视图有 4 个窗口：

· xSOFTip 浏览器——列出 xSOFTip 库中所有可用组件。向“系统配置”窗口添加组件时，“系统信息”窗口将随着应用资源信息不断更新。

· “系统配置”：显示应用程序中的 xSOFTip 组件

· “系统信息”：你所选 xSOFTip 以及最适合你应用程序 xCORE 设备所用的资源。

· “开发者专栏”：关于 xSOFTip 组件的在线信息

每个组件都有一个范围，显示 xSOFTip 组件的状态：

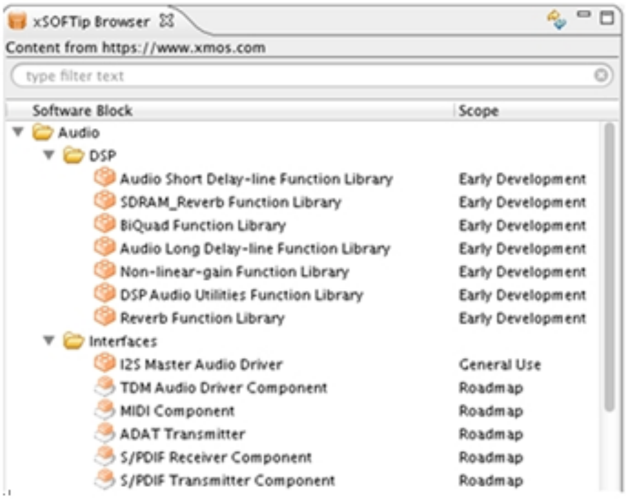


图 7 xSOFTIP 范围

·“一般用途”：xSOFTip 包含一整套自 XMOS 解压的产品。

可使用整套资源信息。我公司尝试各种方法来确保该模块功能正确，但由用户负责使用本模块产品的最终质量。

· 早期开发：xSOFTip 适用于产品开发并能充分发挥功能。然而，必须特别注意验证使用本软件模块的产品。资源信息可用。

· 实验阶段：xSOFTip 现处于实验/原型阶段。代码存在但不完整，资源信息可用。

· 路线图：xSOFTip 位于 XMOS 开发路线图上。xSOFTip 存在估算资源信息，但代码不可用。

· 开放源代码社群：由开放源代码社群研发 xSOFTip，资源信息不可用。

在“浏览器”窗口中选择组件时，“开发者专栏”中就显示了关于组件的信息，包括组件做什么、组件特征及哪种 xKIT 开发套装适用于本 xSOFTip。

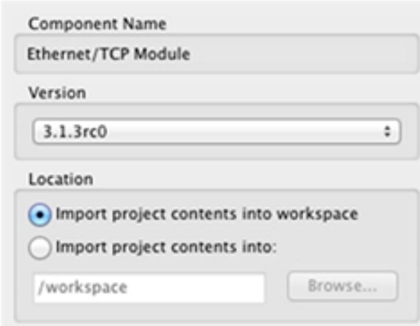
将组件添加至“系统配置”窗口后，点击组件向左箭头，从而在“开发者专栏”显示个性化配置选项的附加信息。

6.1.1 将 xSOFTip 加入项目中

按照下列步骤将 xSOFTip 加入项目中

1. 将 xSOFTip 组件拖拽至系统配置视图中（或双击 xSOFTip 浏览器的组件）

图 8 xSOFTIP 浏览器系统信息窗口

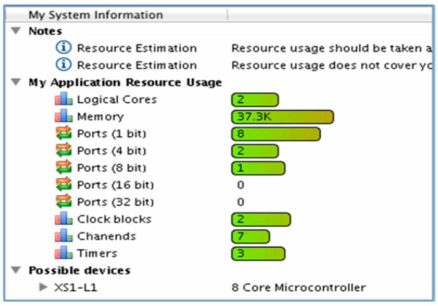


2. 选择意欲导入的组件版本——所有发行版本都可用，默认情况下显示最新版本。
3. 选择意欲添加 xSOFTip 组件的目的项目。
4. 点击“完成”。

6.2 系统信息窗口

“系统信息”窗口显示在“系统配置”窗口选择所需的所有资源。

图 9 xSOFTIP 浏览器系统信息窗口



· 逻辑核心：32 位微控制器核心。XMOS 多核心微控制器包括 8 核、16 核和 32 核设备。

· 端口：XMOS 多核心微控制器输入/输出引脚与端口链接，使软件能够以极低的延迟从引脚处发送、接收信息。端口具有不同的宽度：1 比特端口链接至 1 个输入/输出引脚，4 比特端口链接至 4 个输入/输出引脚。

· 时钟模块：使用时钟模块严格控制输入/输出引脚时间。

· 信道端点：信道端点为 xConnect 系统的一部分，允许核心通过低延迟 xConnect 信道向其他信道发送信息。

· 计时器：软件使用计时器来控制事件发生时间，计时器以 100MHz 频率运行，精确到 10ns。

6.2.1 识别合适的 xCORE 设备

“系统信息”窗口底部显示了一系列可能的设备，显示出适合本应用程序的 xCORE 多核心微控制器。

6.3 xSOFTip 组件配置

一些组件具有可配置选项，只要组件加入系统配置窗口后，这些选项就可以变更。

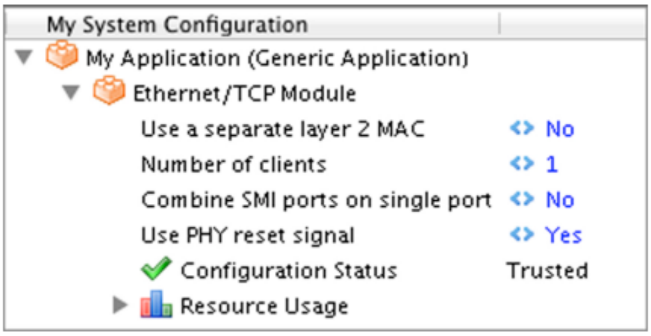


图 10 xSOFTIP 浏览器可配置组件

改变配置时，也应更新“系统信息”窗口中的资源用途。

6.4 从应用程序中生成项目

可以在“系统配置”窗口从组件中自动生成项目。

1. 点击“系统配置”窗口顶部的“生成项目”按钮。
2. 在“生成项目”窗口输入项目名称。
3. 从“目的硬件”列表选择开发板。
4. 点击“完成”

xTIMEcomposer Studio 为你所选择的 xSOFTip 生成项目。

xTIMEcomposer Studio 生成项目的时候会改变至“编辑”视图。

xSOFTip 以 C 代码的形式传送，因此你可以轻易更改它来满足要求并添加现有的 C 函数。

你可以通过“窗口”——“打开视图”菜单随时切换视图。

第 D 部分

编译

内容

 在“项目浏览器”中选择项目，点击“构建”按钮旁边的箭头，选择“调试”或“发布”。

xTIMEcomposer 使用项目中的 Makefiles 来确定编译器中使用的配置设定。

于项目浏览器中双击项目 Makefile 从而将 Makefile 于编辑器中打开，此时可设定编译器选项。XCC 命令行手册（见第 9 节）列出了支持的编译器选项。

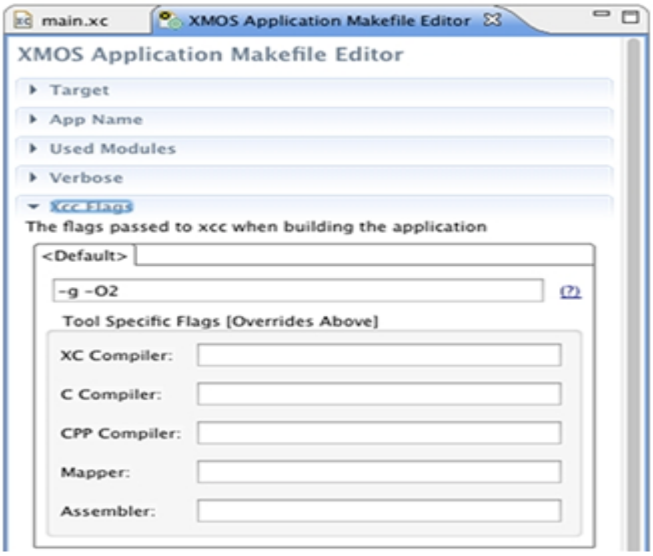


图 11 Makefile 编辑器

如果你的程序中没有错误，则 xTIMEcomposer 将编译的二进制文件添加至项目二进制文件夹。

控制台会报告发生的错误，双击标红的信息，在编辑器中对其定位。

8 XCC 程序指令

xTimeComposer 支持下列指令

#pragma unsafe arrays

（仅适用 XC）本程序禁用运行时的安全检查，从而禁止在下一行动范围内援引无效的数组元素，用于当前函数语句。在函数以外，本程序可用于定义下一个函数。

#pragma loop unroll (n)

（仅适用 XC）本程序控制下一行动的次数，而无论当前函数是否展开。n 指出了显示的迭代数，只在优先级 01 或更高级别才可以展开。省略系数 n 将使编译器完全展开循环。函数之外本程序可以忽略，编译器在不能展开时将会发出警告。

#pragma stackfunction n

本程序为当前编译单元中的下一个函数语句分配 n 个字符（整数）的堆栈空间。

#pragma stackcalls n

（仅适用 XC）本程序为下一个语句中的任意函数分配 n 个字符（整数）的堆栈空间。如果下一语句不包含函数调用，则下一语句将出现在另一个函数中。

#pragma ordered

（仅适用 XC）本程序控制下一选择语句的编译。本选择语句这样编译：选择开始时如有多个事件就绪，则优先选择发生较早的事件。

#pragma select handler

（仅适用 XC）本程序表示下一函数语句为选择处理程序，该选择处理程序可用于选择项中，如下面例子所示。

```
#pragma select handler
void f(chanend c, int &token, int &data);

...
select {
  case f(c, token, data):
    ...
    break;
}
...
```

效用在于启用资源中的事件作为函数的第一个参数。事件发生时，选择处理器会在示例之前发生效用。

该选择处理器第一个参数必须为传输型，而不得为回传型。

如果资源为关联状态，例如为一种境况，则等候事件之前进行的选择不会改变该状态。

#pragma fallthrough

（仅适用 XC）本程序表示下面多分支结构将会在无中断语句或返回语句的情况下落入下一个多分支结构。由于落入过程，本程序将会阻止编译器发生报警或故障。

#pragma xta label "name"

本程序提供标签可用于说明时间限制。

#pragma xta endpoint "name"

（仅适用 XC）本程序规定了终点，常出现在输入或输出语句之前。

#pragma xta call "name"

（仅适用 XC）本程序定义了（函数）调用点标签。使用本程序定义函数调用示例。例如，如果一个函数包含一个回路，则可根据函数调用点将该回路迭代值设定为一个不同的数值。

#pragma xta command "command"

（仅适用 XC）本程序允许在源代码中嵌入 XTA 命令。每当二进制文件载入 XTA 时，所有命令都会运行。命令按照它们在文件中的顺序执行，但不同源文件中的命令顺序并未界定。

#pragma xta loop (integer)

（仅适用 XC）本程序将给定环路 XTA 迭代应用于含程序的回路中。

9 XCC 命令行选项

本章内容：

· 总体选项

· 警告选项

· 调试选项

· 优化选项

· 预处理器选项

· 链接器和映射器选项
· 目录选项

· 影响 XCC 的环境变量

· 由<platform.h>提供的板支撑

XCC 为 xCORE C, C++和 XC 编译器的前面部分。典型应用包括在数据块上进行编码及数据的预处理、编译、组合、链接和映射。一些选项允许在中间阶段终止进程，而另一些选项则会直接进入一个处理阶段。大多数选项都具有否定形式（如-fno-选项）。选项与参数之间允许存在空间。

应用程序 Makefile 中界定了应用程序的设置。于项目浏览器中双击 Makefile 从而将其于 Makefile 编辑器中打开。

9.1 总体选项

编译的四种可能阶段分别为预处理、正确编译、组合和链接/映射。前三个阶段适用于单个源文件，以产生目的文件。链接和映射将目的文件和 XN 文件组合在一个可执行 XE 文件中，其中每个数据块都包含代码和数据段。

source-file　源文件后缀决定其默认处理方式。

拓展名	文件类型	由 XCC 进行预处理
.xc	XC 源代码	Y
.c	C 源代码	Y
.cpp	CPP 源代码（为了兼容，可识别扩展名为 cc, cp, c++, C 和 cxx 的文件）	Y
.S	汇编码	Y

.xta	xCORE 计时分析仪下标	N
.xn	xCORE 网络描述	N
.xi	XC 源代码	N
.i	C 源代码	N
.ii	C++源代码	N
.s	汇编码	N
其他	发送至链接器的目的文件	N

图 12 XCC 可识别的文件扩展名及其含义

-xlanguage 规定了下列输入文件的语言。本选项适用于下一个-x 选项之前的所有输入文件。语言支持值为：

xc

c

c++

assembler

assembler-with-cpp

xta

无（关闭语言规范）

-std=标准

规定了输入 C 或 C++文件所用的不同语言。标准支持值为：

c89

ISO C89

gnu89

扩展名为 GNU 的 ISO C89

c99

ISO C99

gnu99

扩展名为 GNU 的 ISO C99（C 程序默认）

c++98

ISO C++ (1998)

gnu++98

扩展名为 GNU 的 ISO C99（C 程序默认）

-fsubword-select

在 XC 中，允许在目的变量尺寸小于 32 位的信道输入中选择。

该程序为基于 XS1-L 设备的目的变量默认值，而非基于 XS1-G 设备的目的变量默认值。进一步详情，请看§36.3。

-target=platform

规定了目的平台，平台配置必须包含在文件 platform.xn 中，从而按照 XCC_DEVICE_PATH 环境变量（见§9.8）中规定的路径查询配置。

-foverlay

支持存储器迭代；以迭代根目录标示的函数存放在外部存储器中，可在运行时根据需要加载。编译及链接时应跳过选项，应用程序中应包含迭代运行时间。

-foverlay=flash

支持闪存迭代运行时间内的存储器迭代链接。只有从闪存中启动迭代时，才能于数据块上启用迭代。

-foverlay=syscall

支持系统调用迭代运行时间内的存储器迭代链接。从采用系统调用的主机中加载迭代。

-fxscope

启用 XScope 追踪支持。目的变量的 XN 文件必须包括 XScope 链接。编译和链接时传递该选项。

-funroll-loops

使用较少的迭代次数来展开环路，该选项在-O2 及以上版本中启用。

-finline-functions

将简单函数集成在调用器上，该选项在-O2 及以上版本以及-Os 上适用。

-pass-exit-codes

返回编译各阶段产生的数值最大的错误代码（默认情况下返回 1，如果编译器所有阶段都没有成功，则返回 0）

-c

编译或汇编源文件，为每个源文件产生一个目的文件，但不会链接或映射。默认情况下，目的文件文件名将源文件的后缀改为.o（例如 a.c 产生 a.o 的目的文件）。

-S

正确编译后停止，此时每一个未汇编输入文件将产生一个汇编代码文件。默认情况下，汇编文件文件名将源文件的后缀改为.s。
不需要编译的输入文件忽略不计。

-E

只需对源文件进行预处理，将经预处理过的源文件输出为标准输出文档。

不需要预处理的文件忽略不计。

-ofile

在文件中加入输出模块。

如果未指定-o，则为可执行文件添加后缀 a.xe，作为后缀名为.o 源文件的目的文件处，汇编代码文件的源文件后缀名为.s，所有预处理 C/C++/XC 源文件都为标准输出文档。

-v

（以标准误差）打印编译各阶段中执行的命令。同时打印 XCC，预处理器和正确编译器的版本号。

-###

与-v 相同，不执行的命令和引用的命令参数除外。

--help

打印支持命令行的选项描述。如果指定了-v 选项，则同样将--help 传递至 XCC 调用的子进程中。

--version

显示版本号和版权

9.2 警告选项

许多警告都可以通过以-W 开头的选项控制。下列选项都具有以-Wno-开头的否定形式以关闭警告：

-fsyntax-only

出现语法错误时勾选该选项，然后退出。

-w

关闭所有警告信息。

-Wbidirectional-buffered-port

关于输入或输出缓冲端口不合格时发出警告。本警告默认启用。

-Wchar-subscripts

数组下标为 Char 类型时报警。

-Wcomment

评论开始序列/*出现在/*评论中，或者反斜杠换行符位于//评论中时报警，该项为默认项。

-Wimplicit-int

函数语句中未规定类型时报警；在 C 中，函数语句中无传回类型时报警。

-Wmain

主函数类型不带外部链接传回整数时报警；在 XC 中主函数没有零参数时报警。在 C 中主函数没有适当类型的零或二参数时报警。

-Wmissing-braces

联合初始器未加括号时报警。

-Wparentheses

上下文中赋予真值时如果省略括号，或者内部操作员混乱时会报警。

-Wreturn-type

函数为返回型且缺省为整数时，或返回类型不为空值但返回语句未返回任何值时报警。

-Wswitch-default

切换语句不含默认情形时报警。

-Wswitch-fallthrough

（仅适用 XC）含一条以上语句的切换语句中的示例可落入下一示例。

-Wtiming

不符合时间限制时报警，这是默认项。

-Wtiming-syntax

计时脚本中有无效的句法。这是默认的。

-Wunused-function

如果声明静态功能但尚未定义或未使用非在线静态功能，则会出现此警告。

-Wunused-parameter

将功能参数用于非声明用途时会产生此警告。

-Wunused-variable

如果本地变量或非常数静态变量未用于声明用途， 则会产生此警告。

-Wunused

同-Wunused-function、-Wunused-variable 及-Wno-unused-parameter。

-Wall

打开所有上述-W 选项。

Wall 未暗示以下-W... 选项。

-Wextra

-W

打印针对以下方面的外警告消息：

返回数值或不返回数值的函数 （仅 C、C++）。

逗号表达的表达声明或左边侧无不良影响。此警告可以通过将未使用的表达无效以抑制此警告（仅 C、C++）。

无符号的数值与零的比较结果，使用符号<或<=。

存储类说明符（如 static）不是声明中最重要的（仅 C、C++）。

出现比较，如 x<=y<=z（仅 XC）。

函数的返回类型有一个冗余限定词，如 const。

如果还规定了-Wall 或-Wunused，有关未使用的参数的警告。

将有符号数值转变成无符号数值时，有符号和无符号数值的比较可能产生不正确的结果。（如果还规定了-Wno-sign-compare，无警告。）

集合的初始化程序并不初始化所有元素。

使用指定的初始化程序时，覆盖初始化框时不会产生不良影响（仅 C、C++）。

K&R- style 函数中声明的函数参数，无类型说明符（仅 C、C++）。

if 或 else 语句中的空主体（仅 C、C++）。

指令器与整数零比较，使用<、<=、>或>=等表示（仅 C、C++）。

计数器和非计数器均出现在有条件的表达中。（仅 C++）。

非静态参考或非静态 const 计数器和非 计数器均出现在条件表达式中（仅 C++）。

模棱两可的虚拟基底 （仅 C++）。

有声明寄存器的阵列加下标（仅 C++）。

采集声明寄存器的变量的地址（仅 C++）。

在得到的种类副本构造器中未初始化基本种类（仅 C++）。

-Wconversion

如果负整数常数表达含蓄的转化成无符号类型，则会产生此警告。

-Wdiv-by-zero

如果 compile-time 整数被零除，则会产生此警告。这是默认的。

-Wfloat-equal

如果在相等性比较中使用浮点数值，则会产生此警告。

-Wlarger-than-len

如果对象大于定义的 len 字节，则会产生此警告。

-Wpadded

如果结构中含有填充字节，则会产生此警告。（可以重新排列结构中的框以减少填充字节，从而使结构更小。）

-Wreinterpret-alignment

当重新解释导致更大的对齐时，会产生此警告。

-Wshadow

如果本地变量遮蔽另一个本地变量、参数和全局变量或如果内置函数被遮蔽，则会产生此警告。

-Wsign-compare

将有符号的数值转变成无符号的时，如果有符号和无符号数值比较会产生不正确的结果，则会产生此警告。

-Wsystem-headers

打印在系统数据块文件中发现的结构的警告消息。这不是默认的。参见§9.7。

-Wundef

如果在#if 指令中使用未定义的宏，则会产生此警告。

-Werror

将所有警告都当做错误处理。

-Werror=option

将一条警告消息转变成错误。选项应为前缀为-W 的编译程序的警告选项之一。

默认设置为 flag -Werror=timing-syntax 。将此警告转变成错误意味着计时警告（-Wtiming）也错误，反之亦然。

9.3 调试选项

-g

生成调试信息。

-fxta-info

生成与 XTA 一起使用的计时信息。这是默认的。

-fresource-checks

生成可执行代码，以便在资源分配故障时诱捕，从而可以尽早检测出资源错误。

-save-temps

保存中间文件。这些文件放在当前目录中，并根据源文件命名。

-fverbose-asm

产生外的汇编信息，作为中间汇编程序文件中的备注。

-dumpmachine

打印目的机器并退出。

-dumpversion

打印编译程序版本并退出。

-print-multi-lib

打印从 multilib 目录名称到激活其的编译程序开关的映射。目录名称与开关之间使用“;”，且每个开关的开头为“@”，而不是“-”，多个开关之间无空格。

-print-targets

打印编译程序支持的目的平台。目的名称与-target 选项的目的名称对应。

9.4 优化选项

打开优化，使编译程序尝试改善性能 he/或编码大小，然而这样会牺牲汇编时间及调试程序的能力。

-O0

切勿优化，这是默认的。

-O

-O1

优化。尝试缩短执行时间和代码大小，而不执行任何需要大量汇编时间的优化。

-O2

优化更多。这些优化均不涉及 space-speed 权衡。

-O3

优化再多一些。这些优化可能涉及 space- speed 权衡；高性能偏好小代码大小。

-Os

可以优化最小的代码大小。

-fschedule

尝试记录指令以提高性能。这不是任何优化水平默认的。

9.5 预处理器选项

以下选项控制预处理器。

-E　　仅预处理器，然后退出。

-Dname　　预定义名称为定义为 1 的宏。

-Dname=definition

语汇基元化和预处理定义内容，就像在#define 指令中出现的一样。

-Uname　　删除任何之前的 name 定义。

处理-D 和-U 选项以将之显示在命令行上。

-MD

输出到文件，适用于说明源文件依赖性的合适规则。依赖性文件的默认名称根据是否规定了-o 选项进行定义。如果规定了-o，则文件名为后缀为.d 的-o 的参数的基础名称。如果未规定-o，则文件名为后缀为.d 的输入文件的基础名称。文件名可以被-MF 覆盖。

-MMD　　同-MD，除了忽略对系统数据块的依赖性。

-MF file　　规定写入依赖性信息的文件。

-MP

针对每个源文件的依赖性发出虚假目的。每个虚假目的不依赖任何东西。这些虚拟规则变通错误，从而使可以在不更新 Makefile 文件到匹配的条件下删除数据块文件。

-MT file　　规定了依赖性生成发出的规则的目的。

9.6 连接程序和映射程序选项

以下选项控制连接程序/映射程序。

-llibrary

链接时检索库。连接程序依照规定的次序检索并处理库和对象文件。检索的实际库名称为 liblibrary .a。

所检索的目录包括任何规定的带有-L 的。

库为元素为对象文件的档案文件。连接程序扫描档案中的元素，检索参考但并未定义的符号。

-nostartfiles

切勿与系统启动文件链接。

-nodefaultlibs

切勿与系统库链接。

-nostdlib 切勿与系统启动文件或系统库链接。

-s 删除所有符号表并从可执行转移信息。

-default-clkblk clk

使用 clk 作为默认时钟模块。时钟模块可以使用其在<xs1.h>中的名称或其资源编号规定。

启动代码打开默认时钟模块，将之配置为在不分割的条件下关闭基准时钟，并使之处于运行状态。XC 中声明的端口最初连接到默认时钟模块。如果未规定此选项，默认时钟模块设置为 XS1_CLKBLK_REF。

-Wm, option

将 option 作为一个选项传到连接程序/映射程序。如果 option 中含有逗号，则在逗号处将之分割成多个选项。

想要查看全套高级映射程序选项，输入 xmap--help。

-X mapper option

将 option 作为一个选项传到连接程序/映射程序。带有参数 use –X mapper 的选项需要通过两次。

-report 打印资源使用摘要。

9.7 目录选项

以下选项规定了检索数据块文件和库的目录。

-Idir 添加 dir 到将要用于检索数据块文件的目录清单中。

-isystemdir

在所有目录规定为-I 后，在 dir 中检索数据块文件。

将之标记为系统目录。

编译程序抑制系统目录中的数据块文件的警告。

-iquotedir

仅在 dir 中检索带有#include "file" （不带有#include <file>）的数据块文件 ，然后检索-I 规定的所有目录和系统目录。

-Ldir 添加 dir 到将由-I 检索的目录清单中。

9.8 影响 XCC 的环境变量

以下环境变量会影响 XCC 的操作。OS 特殊路径分隔符（Windows 为“;”，Mac 和 Linux 为“：”）分离多路径。

XCC_INCLUDE_PATH
如果规定使用-I，待检索的目录的清单，但是在命令行出现带有-I 的任何路径后。

XCC_XC_INCLUDE_PATH
XCC_C_INCLUDE_PATH
XCC_CPLUS_INCLUDE_PATH
XCC_Assembler_INCLUDE_PATH
这些环境变量中每一个仅在处理提名语言的文件时适用。如果使用-isystem 规定，变量规定了将要检索的目录清单，但是在命令行出现带有-isystem 的任何路径后。

XCC_LIBRARY_PATH
如果规定使用-L，待检索的目录的清单，但是在命令行出现带有-L 的任何路径后。

XCC_DEVICE_PATH
待检索用于设备配置文件的目录清单。

XCC_EXEC_PREFIX
如果设置由编译程序执行的子程序前缀为此环境变量。当前缀与子程序的名称组合时，不得添加分离的目录。执行汇编程序或映射程序时前缀不适用。

XCC_DEFAULT_TARGET
默认目的平台，就像使用-target=规定那样定位。如果未规定带有-target=的目的且未通过 XN 文件，则使用默认目的平台。

9.9 <platform.h>提供的板支持

在汇编程序期间，编译程序生成一个叫做 platform.h 的临时数据块文件，其中含有变量和宏定义（由目的 XN 文件定义），其中包括：

典型 tileref 变量的声明（参见§46.2）。

端口名称的宏定义 （参见§46.4.2）。

10 使用 XMOS Makefiles

XMOS 开发环境创建的项目的结构由 Makefiles 控制。这些 Makefiles 使用程序 xmake（Gnu Make1[1]的端口）执行构建。构建可以从 XDE 或直接从称为 xmake 的命令行直接执行。

你无需理解 Gnu Makefile 语言就可以使用 XMOS 工具开发语言。常用的 XMOS Makefile 可以为项目、应用和模块提供支持。你只需要规定需要构建在项目 Makefiles 和应用 Makefiles 中的应用.

10.1 项目、应用和模块

应用由应用特有的源代码构成，或者由来自常用代码模块的源代码或二进制库组成。在开发应用时，使用工作空间、项目、应用和模块说明工作区。

工作空间

工作空间为几个项目的容器。

项目

项目为可能含有几个应用和模块的目录及其他与特殊项目有关的文件。项目中可能含有特殊板的代码或参考设计，或为含有供其他项目使用的模块的软件组件。

应用

应用指的是含有源文件和 Makefile（构建到单可执行（.xe）文件中）的应用。作为一项公约，应用目录的前缀为 app_。这些应用出现在 XDE 项目浏览器中顶层水平。

模块

模块为含有源文件和/或二进制库的目录。源文件自身并未构建到任何东西中，但是可以借助前缀为 module_的模块目录，供应用。这些模块出现在 XDE 中项目浏览器的顶层水平。

[2] http：//www.gnu.org/software/make/

10.1.1 结构示例

工作空间的结构示例如下所示。

```
sw_avb/
    app_avb_demo1/
    app_avb_demo2/
    module_avb1/
    module_avb2/
    doc/
sc_xtcp/
    module_xtcp/
    module_zeroconf/
sc_ethernet/
    module_ethernet/
```

此工作空间有三个项目：sw_avb、sc_xtcp 和 sc_ethernet。sw_avb 项目含有两个应用，每个应用构建成单独的二元形式。这些应用可以使用项目中模块的源，且可以使用表格来自其自己的项目（module_avb1 和 module_avb2）和其他项目（module_xtcp, module_zeroconf 和 module_ethernet）的模块。

或者，可以使用以下方式构建工作空间：

```
app_avb_demo1/
app_avb_demo2/
module_avb1/
module_avb2/
doc/
module_xtcp/
module_zeroconf/
module_ethernet/
```

在本例中，所有应用和模块均在工作空间的顶层。

10.2 应用 Makefile

每个应用目录均应含有一个叫做 Makefile 的文件，其里面含有常用的 XMOS Makefile。常用 Makefile 控制构建，且默认包含应用目录及其子目录中的所有源文件。应用 Makefile 支持以下变量分配。

XCC_FLAGS[_config]

规定在构建期间到 xcc 的标记。此选项设置特殊构建配置 config 的标记。如果未给出后缀，它为默认构建配置设置标记。

XCC_C_FLAGS[_config]

如果设置了，所有.c 文件的这些标记发送到 xcc 而不是 XCC_FLAGS。此选项设置用于特殊构建配置 config 的标记。如果无后缀，它设置默认构建配置的标记。

XCC_ASM_FLAGS[_config]
如果设置了，所有.s 或.S 文件的这些标记转到 xcc 而不是 XCC_FLAGS 。此选项设置用于特殊构建配置 config 的标记。如果无后缀，它设置默认构建配置的标记。

XCC_MAP_FLAGS[_config]
如果设置了，在最终链接阶段，这些标记转到 xcc 而不是 XCC_FLAGS。此选项设置用于特殊构建配置 config 的标记。如果无后缀，它设置默认构建配置的标记。
XCC_FLAGS_filename
为规定的文件名重写转到 xcc 的标记。此选项重写所有构建配置的标记。

VERBOSE 如果设置为 1，则激活从 make 系统的冗长输出。

SOURCE_DIRS 规定了与汇编内容的应用目录有关的目录清单。默认包含所有目录。

INCLUDE_DIRS 规定待查找的目录，包括在构建期间创建的文件。默认包含所有目录。

LIB_DIRS 规定在构建期间链接到应用的库的目录。默认包含所有目录。

EXCLUDE_FILES 规定未汇编到应用中的源文件名称的空格分隔清单（不包括其路径）。

USED_Modules
规定汇编到应用中的模块目录的空格分隔清单。给出的模块目录中不得有其全部路径，而不论其来自哪个项目，例如：

```
USED_MODULES = module_xtcp module_ethernet
```

MODULE_LIBRARIES
此选项规定了从规定了多个的模块得到的优选库清单。详情参见§11。

10.3 项目 Makefile

不仅每个应用有自己的 Makefile，项目也应有一个顶层 Makefile。此 Makefile 控制项目内应用的构建，并赋予其一个变量以完成此任务：

BUILD_SUBDIRS
规定将要构建的应用目录的空格分隔清单。

10.4 module_build_info 文件

每个模块目录应含有一个名称为 module_build_info 的文件。此文件通知如何在模块中构建文件的应用（如果应用的结构中有模块）。它可以选择含有以下分配变量中的几个。

DEPENDENT_MODULES

规定模块的依赖性。当应用包含一个模块时，它还包括其所有依赖性。

MODULE_XCC_FLAGS

规定在从当前模块汇编源文件时转到 xcc 的选项。此定义可以参考应用 Makefile 的 XCC_FLAGS 变量，例如：

```
MODULE_XCC_FLAGS = $(XCC_FLAGS) -O3
```

MODULE_XCC_XC_FLAGS

如果设置了，此模块中的所有.xc 文件的这些标记转到 xcc 而不是 MODULE_XCC_FLAGS。

MODULE_XCC_C_FLAGS

如果设置了，此模块中的所有.c 文件的这些标记转到 xcc 而不是 MODULE_XCC_FLAGS。

MODULE_XCC_ASM_FLAGS

如果设置了，此模块中的所有.s 或.S 文件的这些标记转到 xcc 而不是 MODULE_XCC_FLAGS。

OPTIONAL_HEADERS

规定特殊数据块文件为可选配置数据块。此数据块文件并不存在在模块中，而是由使用该模块的应用提供。如果此文件中有应用，则构建系统将特殊 macro filename_h_exists 转到 xcc。如果未提供文件，这样可以使模块提供默认配置数值。

11 使用 XMOS Makefiles 创建二进制库

本章内容：
module_build_info 文件
模块 Makefile
使用模块

应用 makefiles 默认使用的模块系统包括源代码水平的常用模块。但是，可以将模块构建至二进制库无需源代码就可以分布。

构建至二进制库的模块需要分成源以构建库和源/包括将要在库中分布的。例如，你可以规定以下结构。

```
module_my_library/
        Makefile
        module_build_info
        libsrc/
           my_library.xc
        src/
           support_fns.xc
        include/
           my_library.h
```

此结构的目的是将源文件 my_library.xc 汇编到库中，且库将分布在 src 沿线，且包括目录（而不是 libsrc 目录）。

11.1 module_build_info 文件

想要构建二进制库时，需要在 module_build_info 文件中设置一些额外变量。必须设置其中一个库或库变量。

LIBRARY　　此变量规定了将要创建的库的名称，例如：

```
LIBRARY = my_library
```

LIBRARIES　设置此变量而不是库变量以规定应构建的几个库（有不同的构建标记），例如：

```
LIBRARY = my_library my_library_debug
```

此清单中的第一个库为在应用包含在此模块中时将要链接的默认库。此应用可以通过添加其名称到其 MODULE_LIBRARIES 清单以规定其他库中的一个。

LIB_XCC_FLAGS_libname

此变量可以设置汇编库 libname 时转到 xcc 的标记。此选项额可用于将不同的汇编标记转到库的不同变量中。

EXPORT_SOURCE_DIRS

此变量应含有未汇编到库中的并作为源分布的目录的清单，例如：

```
EXPORT_SOURCE_DIRS = src include
```

11.2 模块 Makefile

构建到库中的模块可能有一个 Makefile （与普通、仅源模块不同）。此 Makefile 的内容只需要为：

此 Makefile 有两个目的。运行 make all 构建库。调集目的 make export 将创建一份称为输出的目录中的模块的副本，该模块中报告不含有库源。在上述示例中，输出模块如下：

```
export/
  module_my_library/
     module_build_info
     lib/
       xs1b/
         libmy_library.a
       src/
         support_fns.xc
       include/
         my_library.h
```

11.3 使用模块

应用可以依照使用源模块的方法使用库模块（包括 USED_MODULE 清单中的模块名称）。 可以使用会产生相同的最终结果的库源或输出模块。

第 E 部分
计时

内容

12 使用 xTIMEcomposer 给程序计时

本章内容
启动计时分析仪
代码部分计时
规定计时要求
添加程序执行信息
验证汇编期间的计时要求

xCORE 计时分析仪使你能够决定在你的目的平台上执行代码所需要的时间。因为 xCORE 结构的确定性，这些工具可以测试执行一部分代码的最短和最长时间。在与用户规定要求组合时，这些工具可以在汇编时间确定，是否所有代码的计时重要部分均由在其截止时间内执行的保证。

12.1 启动计时分析仪

依照以下步骤，在计时分析仪等控制下加载程序：

1. 在项目浏览器中选择一个项目。
2. 选择运行时间配置。
3. 在左边的面板中，双击 XCore Application。xTIMEcomposer 创建了新的结构并在右边面板中显示默认设置。
4. xTIMEcomposer 尝试识别目的项目可执行项目。需要选择其中一个时，单击项目文本框 的 Browse，并在项目 Selection 对话框中选择项目。然后，单击 Search Project 和选择 Program Selection 对话框中的可执行项目。

你必须已经汇编了你的程序，而无任何可执行错误以供选择。

5. 在 Name 文本框中输入配置的名称。
6. 保存配置并启动计时分析仪，单击 Time。xTIMEcomposer 加载你的程序到计时分析仪中并在 Timing 透视图中打开。在此透视图中编辑器为只读的，目的是确保二元和源代码之间的关系保持一致。

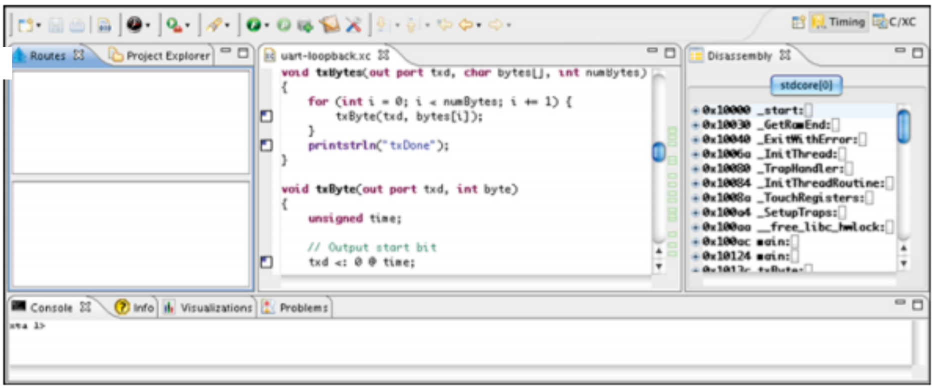


图 13：
计时透视图

xTIMEcomposer 记住最后一次加载程序使用的配置。之后想要使用相同的设置加载 XTA 程序，则只需单击 XTA 按钮。想要使用不同的配置时，单击 XTA 按钮右边的箭头并从下拉清单中选择一个配置。

12.2 代码部分计时

路径由一套控制可以在程序中两点（或端点）之间流动的所有路径构成。每个路径有一个最佳案例时间，也即支路一直走执行时间最短的路线，对应最差案例时间。

依照下面的步骤规定路径并分析它：

1. 右击编辑器边缘的端点标记，并选择 Set from endpoint。xTIMEcomposer 的标记的右上四分之一显示一个绿色的点。

2. 右击一个端点标记并选择 Set to endpoint。xTIMEcomposer 的标记的右下四分之一显示一个红色的点。

你可以在端点上面规定一个起点。你还可以在端点处或其下面规定一个起点，定义；流式输出路径，然后返回到功能。这是称为 multiple times 或 from within a loop 的函数。

3. 单击主工具栏中的 Analyze Endpoints 按钮。xTIMEcomposer 分析规定的路径中的所有路线，在 Routes 视图的下面使用树状展示，并在 Visualizations 视图的 Structure 标签使用图形展示。

或者，为分析函数执行所耗时间，点击主工具栏上的“分析函数”按钮，从下拉列表中选择一个函数。

xTIMEcomposer 提供在汇编期间会保留次序的所有描述的端点标记。这些描述包括输入/输出 操作和函数调集。

12.2.1 可视化路径

Routes 视图显示路径的结构展示。每次分析路径时，向顶部面板添加一个输入。单击路径以在底部面板查看它。使用以下节点展示它：

源代码级函数。

依照顺序执行的节点的清单。

一套有条件执行的节点。

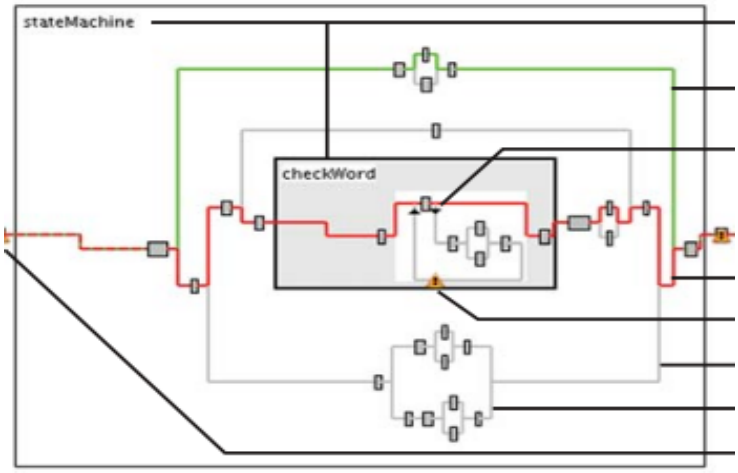
包含一系列节点的回路，其中最后一个节点可以从支路返回第一个节点。

含有指令的直线顺序的模块。

单一机器指令。

12.2.2 可视化视图

可视化视图提供了路径的图形展示。Structure 标签使用从左到右的线条表示路径，如下面的例子所示。在代码分支时路径插入多路径，其所有路线的末端汇合。最佳案例计时路线使用绿色高亮表示，最差案例路线使用红色高亮表示，而所有其他路线均用灰色表示。



函数名称

最佳路线

回路

末端
最差路线
未知路线
指令模块
开始

图 14 可视化视图

图 节 质 在 构视 中 标放在 点 时性 的总结。单击一个节点以高亮显示其在编辑器中的源代码，或双击进入节点起点的线条。在结构视图中，双击函数名称以扩展或收缩它。

12.3 规定计时要求

计时要求规定了路径中的路线让程序正确执行需要执行的时间。在路径视图的顶部面板中，每条路径的状态使用其名称左边的图标表示：

计时要求规定了路径中的路线让程序正确执行需要执行的时间。在路径视图的顶部面板中，每条路径的状态使用其名称左边的图标表示：

? 未规定计时要求。

✔ 规定并满足了计时要求。

✔ 规定并满足了计时要求，然而需要等待所有输入/输出指令准备就绪以执行。

✖ 规定然而并未满足计时要求。

右击路径并选择 Set timing requirements 一个对话框将打开，以规定计时要求。输入路线的最长执行时间（如 ns、cycles 或 MHz）并单击 OK。xTIMEcomposer 更新路径的状态。

12.4 添加程序执行信息

在一些条件下，无额外信息时计时分析仪无法证明计时。常见的条件的示例包括：

路径中含有可以暂停未知时间长度的输入/输出指令。

路径中含有数据相关退出条件的回路。

路线未满足计时要求，但是该路线仅因为错误条件而执行，因此对计时并不重要。

在这些情况下，你可以提供有关你的程序执行的及时分析仪的额外信息。有了此额外信息，分析仪可以证明满足路线的计时要求。你呢可以提供的信息包括：

回路迭代的次数：右击回路节点并选择“设定环路迭代”以显示一个对话框。输入最大回路计数并单击 OK。

输入/输出指令的最长暂停时间：右击指令节点并选择“设定指令时间”以显示一个对话框。输入一个数值，选择时间/速率单位（如纳秒或兆赫）并单击 OK。

排除路径中的一条路线： 右击节点并选择“排除”。

12.4.1 细化最差案例分析

默认计时分析仪认为路径始终遵循执行时间最长的支路。如果你知道不是这样，例如在模拟期间检查或对你的程序进行正式的分析，你可以改善分析仪所使用的参数。你可以作出的改善包括：

规定函数调集的绝对执行时间： 右击函数节点并选择“设定函数时间”以打开一个对话框并单击 OK。

规定路线的绝对时间：按下 Ctrl 不放（Windows，Linux）或 （Mac）不放并单击两个指令节点，然后右击并选择 Set path time 打开一个对话框。输入时间并单击 OK。

规定节点执行的次数：默认分析仪假定节点的执行次数为其范围中每个回路计数的乘积。为了将迭代次数转变成一个绝对的数值，右击节点并选择 Set loop scope 打开一个对话框。选择 Make scope absolute 并单击 OK。

规定回路中某条件的执行次数：默认分析仪假定有条件的节点始终遵循执行时间最长的路线。为了规定执行有条件的目的的次数，右击目的节点并选择 Set loop path 迭代以打开一个对话框。输入迭代次数并单击 OK。

12.5 编译时验证计时要求

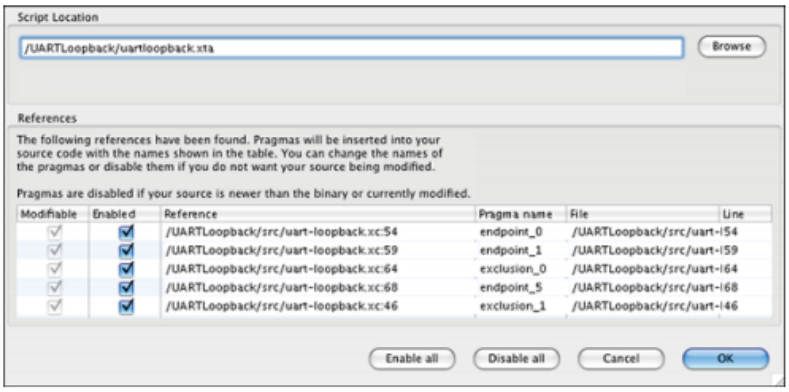
在规定了程序的计时要求后（包括有关其执行的任何改善），你可以生成一个在汇编时间检查这些要求的脚本。

依照这些步骤，创建检查“路径”视图中规定的所有计时要求的脚本：

1. 单击“生成脚本”按钮。
2. 在“脚本位置”文本框中输入脚本的文件名。文件的扩展名必须为.xta。
3. 在“程序名”框中修订其数值，以变更添加到源文件的编译指令的名称。
4. 单击 OK 以保存脚本并更新你的源代码。xTIMEcomposer 添加脚本到你的项目并在编辑器中打开它。它还使用脚本需要的任何编译指令，更新你的源文件。

下次编译程序时，检查计时要求，并在发生任何故障时报告为汇编错误。双击计时错误以查看脚本中的故障要求。

图 15：
脚本选项对话框



13 自命令行使用 XTA

本章内容
常用命令
查看结果
细化计时结果
横须结构
进程自动化
通过 Jytbon 界面为 XTA 编写脚本

可于 xTIMEcomposer Studio.命令行或控制台上交互使用 XTA 工具。

13.1 常用命令

本节概述了大量的常用命令，可自命令行运行。

13.1.1 加载二进制库

加载库类型。

```
load  <FILE  NAME>.
```

13.1.2 路由

路由是代码中时间关键部分。它由一组所有路径组成，而控制可以通过这些路径在程序中的两点（端点）之间流动。可以通过函数计时、回路安全计时或两个端点之间计时，创建路由。

13.1.3 端点

端点指的是在编译过程期间，必须保护的任何源线，且必须保持其相对其他端点的次序。
输入以下内容，显示一份所有端点列表：

```
list allendpoints
```

如果根据集合代码规定路由，则可以在端点使用任何有效的标签/程序计数器（PC）。但是，程序计数器可以分类为不可移动端点，因为在不同的编译中，它们也不同，因此不鼓励在脚本中使用它们。

13.1.4 添加端点到源中

源线可以标记上端点编译器指令，以确保该端点可移动。例如，图16示出了带有端点编译器指令注解的，称为启动和停止的函数。

```
int g(in port p) {
    int x, y;

    # pragma xta endpoint " start "
    p :> x;

    # pragma xta endpoint " stop "
    p :> y;

    return (y - x);
}
```

图16：将端点编译器指令放入源中。

示出一份端点类型的列表：

```
list endpoints
```

13.1.5 端点之间的计时

端点类型之间的计时：

```
analyze endpoints <from ENDPOINT> <to ENDPOINT>
```

XTA不给经过多个xCORE窗口并联的代码计时，因此链各个端点必须在相同的窗口并联上。

一项分析可能导致生成多个路由。

13.1.6 计时函数

将函数名称输入控制台：

```
analyze function <FUNCTION>
```

这将创建路由，该路由说明了从函数输入点到所有函数返回点的一套所有可能的路径。

13.1.7 计时回路

回路类型计时：

```
analyze loop <ANY>
```

这创建说明通过该回路的所有可能的路径路由。如果它是开始和终止端点相同的端点之间的计时高效捷径，则该点在回路中，且规定了排斥，从而排除在回路外的所有东西。

一项分析可能导致生成多个路由。

13.1.8 设置计时要求

定义路由类型的计时要求：

```
set required <route id> <value> <MODE>
```

所支持的计时模式定义在§14.3中。输入以下内容，就可以发现路由ID：

```
print summary
```

或者，在指令行或脚本中使用-字符，以查看上次分析的路由。

13.2 查看结果

13.2.1 路由标识

所有经分析的路由给出一个唯一的路由标识。但是，当参见脚本中的路由时，使用该路由ID不一定会产生可移动或可靠的脚本。

在很多情况下，需要引用的唯一路由是上次分析的。将“-”字符作为路由ID，就可以实现此目的。如果最后一条指令创建多个路由，则“-”字符指的是所创建的所有路由。

13.2.2 节点标识

在单一路由中，所有节点均被赋予一个唯一的ID号。这是一些控制台指令需要的输入。在此情况下，可以使用字符“-”以查看路由的顶级节点。

13.2.3 概要

显示一份所有路由类型的列表：

```
print summary
```

使用以下指令以显示特定路由的详情：

```
print routeinfo <route id>
```

13.2.3.1 违例

当设置了路由的计时要求，且该路由执行所用时间长于要求的时间，则该时间差称为违例。此数值规定了在保证满足计时要求的情况下，路由执行速度可以快多少。

13.2.3.2 松弛

当设置了路由的计时要求，且该路由执行所用时间短于要求的时间，则该时间差称为松弛。此数值规定了在保证满足计时要求的情况下，路由执行速度可以慢多少。

13.2.4 结构

以xTIMEcomposer Studio类型，显示路由的结构：

```
print structure <route id>
```

XTA使用的结构如§13.4.2所述。

13.2.5 源代码注解

显示由路由类型执行的源代码：

```
print src <route id>
```

如果只能使用部分路由，则可以规定节点ID：

```
print src <route id> <node id>
```

13.2.6 指令跟踪

为了帮助开发人员理解路由的执行流，则XTA可以创建代表性指令跟踪。类型：

```
print trace <route id>
```

跟踪时会显示回路，从而跟踪可能变得非常大。可以在任何时间，按下指令行工具中CTRL+C，撤销跟踪操作。
输入以下内容，将跟踪重新导向文件：

```
print trace <route id> > <file>
```

默认打印最差情况路径的跟踪。输入以下内容，就可以变更为打印最佳情况路径：

```
config case best
```

13.2.7 取停止操作指令
可能需要xCORE设备，以在从内存取较多指令时，在特定时间终止。这可能产生停止操作指令问题。硬件发生这种情况时，在跟踪中显示为FNOP。
在xTIMEcomposer Studio中，在发生时将之插入反汇编中。

13.2.8　缩放结果
默认XTA缩放所有计时结果。这意味着使用合适的单位（ms、us、ns）打印时间数值。输入以下内容，可以变更此设置，从而使所有时间的打印单位都为ns：

```
config scale false
```

13.2.9 未知事件
如果无法确定回路迭代计数及指令的执行时间，则XTA可能无法确定某节代码的精确计时。输入以下内容，这些未知条件可能显示在控制台上：

```
list unknowns <route id>
```

§13.3说明了解决这些警告的方法。

13.3 改善计时结果
可能出现XTA无法完全确定某节代码的计时的情况，例如因为无法确定回路计数。这可以通过添加定义解决。有两种添加定义的方式：到全局列表，或到路由特定列表。在创建时，这些将被添加到用于每个路由的全局列表中。

⚠全局列表的使用将产生更加简洁的脚本。但是，重要的是应谨慎处理添加到全局列表中的定义，因为如果无法用于路由，则它们将被忽略。这允许在创建任何路由前，创建全套定义，但是这并不意味着这些定义中的错误可能被错过。路由特定定义（在路由创建后添加）会标记出所有错误。

13.3.1 排斥
并非路由中的所有执行路径都是计时重要的。路由中可能含有计时代码不重要的错误的情况。通过添加排斥，可以忽略计时脚本中的这些路径。排斥告诉XTA忽略经过这些代码点的所有路径。排斥能够添加到全局列表中，或用于特定路由。

在现有路由类型上设置一个排斥：

```
set exclusion <route id> <ANY>
```

在路由创建期间，输入以下内容，添加排斥到排斥列表中：

```
add exclusion <ANY>
```

输入以下内容，列出排斥的全局列表：

```
list exclusions
```

从全局列表类型中删除排斥：

```
remove exclusion <ANY|*>
```

例如，考虑图17中的代码。

```
int calculate ( int a, int b) {
    if ( willOverflow (a, b) {
        # pragma xta label " overflow "
        return processOverflow ();
    }
    return a + b;
}
```

图17：排除一个有效路径

在给计算函数计时时，忽略错误情况：

· 使用路由特定定义：
· 分析函数计算
· 设置排斥 -溢流

使用全局定义：

· 添加排斥溢流
· 分析函数计算

尽管功能等价，通过全局定义机制的排斥可能使路由创建更快且内存效率更高。这是因为，在路由创建期间可以考虑全局排斥，从而使检索空间变小。如果是路由创建后排斥，则在任何删减前，创建整个路由。

13.3.2 回路迭代

回路迭代计数无法知晓。如果可能，编译器将回路迭代计数告知XTA。但是，无法通过静态方式知道一些回路计数。在这些情况下，开发人员必须说明最差数值。
编译器未发出任何回路迭代计数，除非激活优化（-O1或更大）。
一些回路为自循环的（回路主体与标题相同），且因此，迭代计数最小值为1。
在现有路由类型上设置回路迭代：

```
set loop <route id> <ANY> <iterations>
```

输入以下内容，添加迭代计数到在路由创建期间将要使用的迭代计数列表中：

```
add loop <ANY> <iterations>
```

输入以下内容，列出当前全局回路迭代计数类型：

```
list loops
```

输入以下内容，从全局列表类型中删除回路迭代计数：

```
remove loop <ANY|*>
```

例如，考虑图18中的代码。给测试函数计时：

· 使用路由特定定义：

  · 分析函数测试
  · 设置回路 - delay_loop 10

· 使用全局定义：

```
void delay ( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " delay_loop "
        delay_us (1);
    }
}

int test () {
    delay (10);
}
```

图18：设置回路迭代

· 添加回路delay_loop 10
· 分析函数测试

13.3.3 回路路径迭代

一个回路中可能含有多条路径。设置回路迭代后，工具假定所有迭代将采用经过回路的最差执行路径。然而，情况并不总是这样，规定通过回路的每条路径的迭代次数，就可以确定更加现实的最差情况。

输入以下内容，在现有路由上设置回路路径迭代：

```
set looppath <route id> <ANY> <iterations>
```

输入以下内容，向将要在回路创建期间使用的回路路径计数中添加一个回路路径计数：

```
add looppath <ANY> <iterations>
```

输入以下内容，显示全局回路路径计数的当前列表：

```
list looppaths
```

输入以下内容，删除全局列表中的一个回路路径：

```
remove looppath <ANY|*>
```

⚠在设置回路路径迭代时需要遵循一些规则：

· 在嵌套回路中，需要先设置外回路迭代。
· 回路路径迭代设置必须小于或等于闭合回路上的回路迭代设置。

如果回路路径迭代设置低于闭合回路的，则回路中必须有无此迭代设置的另一条路径，可以将剩余迭代分配给它。

例如，考虑图19中的代码：

```
void f( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " f_loop "
        if ((i & 1) == 0) {
            # pragma xta label " f_if "
            g ();
        }
    }
}

int test () {
    f (10);
}
```

图19：设置回路路径迭代

给测试函数计时：

· 使用路由特定定义：
    · 分析函数测试
    · 设置回路 - f_loop 10
    · 设置回路路径 - f_if 5
· 使用全局定义：
    · 添加回路 f_loop 10
    · 添加回路路径 f_if 5
    · 分析函数测试

13.3.4 回路范围

默认方式为，XTA假定回路迭代是相对的 – 内回路的迭代将乘以闭合回路的迭代。但是，这不足以描述所有回路结构。如果此假定不正确，则回路计数可以设置为绝对。绝对回路上的迭代计数设置无需乘以闭合回路上设置的迭代。

输入以下内容，设置现有路由上的回路范围：

```
set loopscope <route id> <ANY> <absolute|relative>
```

输入以下内容，添加一个回路范围到将在路由创建期间使用的回路范围列表：

```
add loopscope <ANY> <absolute|relative>
```

输入以下内容，显示全局回路范围的当前列表：

```
list loopscopes
```

输入以下内容，从全局列表中删除回路范围：

```
remove loopscope <ANY|*>
```

例如，考虑图20中的代码：

```
void f( int l) {
    for ( unsigned int i = 0; i < l; ++i) {
        # pragma xta label " outer_loop "
        for ( unsigned int j = 0; j < i; ++j) {
            # pragma xta label " inner_loop "
            g ();
        }
    }
}

void test () {
    f (10);
}
```

图20：设置回路范围

给测试函数计时：

· 使用路由特定定义：

· 分析函数测试

· 设置回路 - outer_loop 10

· 设置回路 - inner_loop 45

· 设置回路范围 - inner_loop absolute

· 使用全局定义：

· 添加回路 outer_loop 10

· 添加回路 inner_loop 45

· 添加回路范围 inner_loop absolute

· 分析函数测试

13.3.5 指令执行时间

一些指令可以暂停处理器。默认的为，XTA报告假定无指令暂停时的计时，而是将它们标记为警告。开发人员必须说明指令的最差执行时间是什么。

输入以下内容，在现有路由上设置指令执行时间：

```
set instructiontime <route id> <ENDPOINT> <value> <MODE>
```

输入以下内容，添加一个指令执行时间到指令执行时间列表中，以在路由创建期间使用：

```
add instructiontime <ENDPOINT> <value> <MODE>
```

输入以下内容，显示全局指令执行时间的当前列表：

```
list instructiontimes
```

输入以下内容，从全局列表中删除一个指令执行时间：

```
remove instructiontime <ANY|*>
```

例如，考虑图21中的代码。

```
void f( port p) {
    # pragma endpoint " instr "
    p :> value ;
}
```

图21：设置一个指令执行时间

给f函数计时：
· 使用路由特定定义：
· 分析函数 f
· 设置指令执行时间 - instr 100.0 ns
· 使用全局定义：
· 添加指令执行时间 instr 100.0 ns
· 分析函数 f

13.3.6 函数执行时间

在一些情况下，有必要定义执行整个函数需要的时间。XTA支持定义函数执行时间。定义函数执行时间后，其中所有未知事件均被忽略，且跨越此函数的任何路由都将使用定义的执行时间，而不是计算它。

输入以下内容，在现有路由上设置一个函数执行时间：

```
set functiontime <route id> <FUNCTION> <value> <MODE>
```

输入以下内容，在函数执行时间列表中添加一个函数执行时间，以在路由创建期间使用：

```
add functiontime <FUNCTION> <value> <MODE>
```

输入以下内容，以显示全局函数执行时间的当前列表：

```
list functiontimes
```

输入以下内容，从全局列表中删除一个函数执行时间：

```
remove functiontime <FUNCTION|*>
```

例如, 考虑图22中的代码。
输入以下内容，在现有路由上设置一个路径执行时间：

```
set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

输入以下内容，添加一个路径执行时间到路径时间列表中，以在路由创建期间使用：

```
add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

输入以下内容，显示全局路径执行时间的当前列表：

```
list pathtimes
```

输入以下内容，从全局列表中删除一个路径执行时间：

```
remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>
```

例如, 考虑图23中的代码。

```
int f() {
    int time ;
    timer t;
    # pragma xta endpoint " start "
    t :> time ;
    # pragma xta endpoint " stop "
    t when timerafter ( time + 100) :> time ;
}

void test () {
    f ();
}
```

图23：设置一个路径执行时间

给测试函数计时：
· 使用路由特定定义：
· 分析函数测试
· 设置路径执行时间 - start stop 1000.0 ns
· 使用全局定义：
· 添加路径执行时间 start stop 1000.0 ns
· 分析函数测试

```
void delayOneSecond () {
    g ();
    }

void test () {
    delayOneSecond ();
}
```

图22：设置一个函数执行时间

给测试函数计时：
· 使用路由特定定义：
· 分析函数测试

· 设置函数执行时间 - delayOneSecond 1000.0 ms
· 使用全局定义：
· 添加函数执行时间 delayOneSecond 1000.0 ms
· 分析函数测试

13.3.7 路径执行时间
在一些情况下，有必要定义执行特定节代码需要的时间。在此情况下，XTA支持定义路径执行时间。定义路径执行时间后，其中所有未知事件被忽略，且跨越此函数的任何路由都将使用定义的执行时间，而不是计算它。
13.3.8 活动窗口并联
默认方式为，XTA找到程序内所有窗口并联上的路由。但是，它可能限制XTA，使其只能在程序中窗口并联的子设置上工作。适用的所有指令的窗口并联设置称为活动窗口并联。
输入以下内容，选择活动的窗口并联：
· 添加窗口并联 <窗口并联 id>
· 删除窗口并联 <窗口并联 id|*>
· 列出窗口并联

13.3.9 节点频率
xCORE设备由大量节点组成，每个节点由一定量的xCORE窗口并联组成。节点运行的频率被使用二进制代码定义，且XTA将之读出，并在加载二进制代码时，配置节点频率。可以通过实验确定在不同的频率下将发生什么情况（如有需要）。
输入以下内容，变更节点的频率：
```
config freq <node id> <tile frequency>
```

13.3.10 逻辑核心的数量
在编译时间知道在窗口并联上运行的逻辑核心的最大数量，且XTA从每个窗口并联的二进制代码中提取此信息。可以通过实验确定如果使用不同的核心运行将得到什么结果（如有需要）。
输入以下内容，以变更节点/标题的核心数量：
```
config cores <tile id> <num cores>
```
13.4 程序结构
程序写入多个源文件中，每个源文件中含有函数。每个函数将含有陈述、回路（如 目的/时间/ 行动）、条件（如如果/开关）及函数调用的次序。
13.4.1 编译XTA
编译器输出信息使XTA能够将源和指令关联起来。此信息为默认的，但是可以通过添加以下标记到编译器选项中将之失活：
```
-fno-xta-info
```
编译器还支持在不影响优化的情况下，添加除错信息。XTA分析代码无需除错信息，但是无除错信息，就无法实现指令和源代码之间的映射。为了添加除错信息，使用以下程序编译：
```
-g
```

13.4.2 结构节点

编译器工具使用xCORE窗口并联的一个程序，创建一个二进制文件。XTA使用该二进制文件生成准确的计时结果。

创建路由后，XTA分析创建结构（非常接近的表达高级程序结构）的二进制。它将程序分解成结构节点，并以树的结构显示。

然后，计算每个结构节点的最差和最佳事件的执行时间。

此计算所采用的方式取决于结构节点的类型。整个路由的最差和最佳事件执行时间为子节点的最差和最佳事件执行时间的总和。

结构节点可以为以下类型：

· 指令：程序最基础的结构模块是指令。

· 模块：一个无有条件分支的指令节点的列表，因此可以依照次序执行。模块的最差/最佳事件执行时间为其组件指令的加和。

· 次序：一份依照次序执行的结构节点的列表。次序的最差/最佳事件执行时间为其子节点的最差/最佳事件执行时间的总和。

· 有条件：一套结构节点，最多执行这些节点中的一个节点。如果此节点在回路中，则可以选择每个节点执行一次迭代。在一些情况下，整个条件都是可选的。在该等情况下，最佳执行时间是未选择任何选项。有条件的最差/最佳事件执行时间是通过其每个子节点的最差/最佳事件执行时间确定的。

· 回路：由标题和主体组成（这两个都是结构节点）。标题对应于回路中的有条件测试部分，而主体对应于采用该回路时，所执行的代码。这基本与高水平代码结构对应，如循环或回路。每次迭代执行一次主体。标题执行次数比迭代次数多一。回路的最差/最佳事件执行时间为其标题的最差/最佳事件执行时间乘以（迭代次数 ＋ 1），加上主体的最差/最佳事件执行时间乘以迭代次数。

· 自循环： 标题和主体相同的回路。因此，视为其最低回路计数为1。这大致对应于高级代码结构，如执行回路。自循环的最差/最佳事件执行时间为主体的最差/最佳事件执行时间乘以迭代次数。

· 函数： 为函数的高级结构，由其他结构节点的列表组成。函数的最差/最佳事件执行时间的计算方式与次序的相同。

13.4.3 识别节点：代码引用

代码引用是一种规定应用中特殊位置的方式。代码引用由基础和可选回溯组成。基础由引用类型组成，而回溯由引用类型的逗号分离列表组成。

有大量不同的引用类型，所有这些都可以映射到一个或多个指令程序计数器（PCs）中。这一般为一个PC，但是也可以为多个，因为编译器优化或因为用户已经明确给多个指令一个相同的引用。编译器优化（如代码嵌入或展开）将导致相同的引用映射到多个PC中。

不同的引用类型的详细说明如下。依照类型，详细说明当前可执行的实例上传到控制台中的指令。

· 源文件行引用对编译器被定义为属于源代码级基础块的源线有效。使用以下指令，可以在控制台中列出有效行：

```
list   allsrclabels
```

· 使用#编译器指令xta标签，将源标签添加到源代码中。输入以下内容，在控制台中列出

源标签：
list　srclabels
·　调集文件行引用对映射到函数调用的源线有效。输入以下内容，在控制台中列出有效源线：
list　allcalls
·　使用#编译器指令xta调用，将调用标签添加到源代码中。输入以下内容，在控制台中列出源标签：
list calls
·　端点文件行引用可用于映射到有效端点的源线。输入以下内容，在控制台中列出端点：
list　all endpoints

使用#编译器指令xta端点，将端点标签添加到源中。它们必须在行上，然后才能进行输入/输出操作。输入以下内容，在控制台中列出端点：

```
list endpoints
```

· 标签指的引用任何源代码或集合标签的任意文本串。输入以下内容，在控制台中列出标签：

```
list labels
```

集合中的标签必须在可执行的节中。
·　函数指的是包含在二进制中的函数。输入以下内容，在控制台中列出标签：

```
list functions
```

汇编器中的函数必须标记.type指令，以被XTA正确检测出XTA（参见xTIMEcomposer Studio用户指南）。它们还必须在可执行节中。
·　程序计数器（PC）为最低水平引用，给出了十六进制程序计数器数值，起始数据为0x。它们必须映射到程序可执行部分中的指令的PC上。

13.4.4 引用类
特定控制台指令仅对特定类型的引用有效。为特定指令定义的引用类型集合被称为引用分类。
·　端点：可用于计时的引用。这指的是汇编器中的任何引用（PC/标签），且只有映射到行中的源引用才能可靠的用于计时。编译器优化无法清除它们或给其重新排序。在XC代码中，这些对应于带有I/O操作的源线。下面的控制台指令列出了每类中的可用类型：

```
help ENDPOINT
```

·　调集：映射到函数调用的引用。这些被用于回溯中，以识别唯一的代码引用。下面的控制台指令列出了该类可用的类型：
help　CALL
·　函数：映射到函数的引用。以下控制台指令列出了该类可用的类型：

```
help CALL
```

·　标签：以下控制台指令列出了该类可用的类型：

```
help FUNCTION
```

·　PC：以下控制台指令列出了该类可用的类型：

```
help PC
```

可以有一个能够映射到多类型的代码引用。例如，可以有一个名称与程序中函数相同的端点。回溯中引用匹配的方式取决于引用的类型。为了解决此潜在不明确性，可以在此类型前面加前缀的方式，使代码引用为特定类型。

13.4.5 回溯

在一个程序中，代码引用库可以多次发生。例如，可以从多个地方调集函数。引用的回溯是一种限制引用特定事件的方式。考虑图24中所示的实例文件。

```
1  void delay_n_seconds ( int j) {
2      for ( unsigned int i = 0; i < j; ++i) {
3          # pragma xta label " delay_loop "
4          delay_1_second ();
5      }
6  }
7
8  int test () {
9      # pragma xta call " delay_1 "
10     delay_n_seconds (10);
11     # pragma xta call " delay_2 "
12     delay_n_seconds (20);
13     return 0;
14 }
```

图24：使用回溯。

下面的指令可用于给测试函数计时：

· 分析函数测试
· 设置回路 - delay_loop 10

这样可以设置delay_n_seconds到10的两种情况的回路迭代次数。但是，因为作为delay_n_seconds参数经过的迭代次数不同，每次调集时此数值也不同。

为了给测试正确计时，需要给每个回路迭代规定一个不同的时间。这可以通过使用调集引用和回溯实现。

例如：

· 分析函数测试
· 设置回路 - delay_1,delay_loop 10
· 设置回路 - delay_2,delay_loop 20

这告诉工具在从delay_1调集时，设置delay_loop为10次迭代，当从delay_2调集时，设置为20次迭代。上述案例使用的引用有类型源标签的基础引用及回溯或类型调集标签的大小等组成。上述目的还可以通过使用文件行当量实现。

例如：

· 分析函数测试
· 设置回路 - source.xc：10,source.xc：3 10
· 设置回路 - source.xc：12,source.xc：3 20

但是，这不会导致可移动和可靠的脚本实施，因此不鼓励以这种方式使用脚本中的文件行引用。

⚠ 当编译器内联一些代码时（例如上述delay_n_seconds 函数），则一些引用不再有效。在这种情况下，以下引用可能不存在，因为调集不存在：

```
source.xc:10,source.xc:3
```

但是，如果调集可以使用调集标签，则编译器确保即便代码内联，该引用仍有效。因此，在上述案例中，以下引用将仍有效：

```
delay_1,delay_loop
```

13.4.6 引用的范围

引用可以为全局或局部范围。全局范围引用指的是适用于（或得到解决的）全局树。而全局树指的是整个程序的抽象结构声明，之后分析任何路由。局部范围引用指的是适用于（或得到解决的）用户创建路由树的。特定引用是否是局部范围的全局取决于正在执行的指令。以下指令使用全局范围引用：

· 分析路径
· 分析函数
· 分析回路
· 添加排斥
· 添加分支

以下指令使用局部范围引用：

· 设置/添加回路
· 设置/添加回路路径

· 设置/添加回路范围
· 设置/添加指令执行时间
· 设置/添加路径时间
· 设置/添加函数时间

一般情况下，全局范围引用可能导致创建多个路由。

13.5自动化过程

使用脚本使XTA自动化，以确保新应用版本符合计时要求。

13.5.1撰写一个脚本

此脚本文件为XTA控制台指令的次序。每个次序一行。任何开头为#符号的行被视为是一个评论。

开发人员必须将编译器指令插入源代码中，这是使脚本可移动需要的。如果脚本创建过程修订源（如通过插入编译器指令），则必须重建相关二进制，然后才能成功执行该脚本。

建议不在脚本中输入加载或退出指令。应在调集脚本时完成这些指令。

XTA脚本必须使用.xta作为扩展名，以由编译器使用，并得到xTIMEcomposer Studio正确理解。

13.5.2 运行脚本

必须以不同的方式运行脚本，在xTIMEcomposer Studio中或在指令行中。

· 在编译期间：在命令行中，必须手动使.xta脚本经过编译器。默认方式为，计时故障被处理为警告，且脚本中的语法错误被视为是错误。

添加以下内容到编译器参数中，以将计时故障处理为错误：

```
-Werror=timing
```

添加以下内容到编译器参数中，以处理脚本语法错误为警告：

```
-Wno-error=timing-syntax
```

· 批处理模式：在批处理模式中，XTA采集指令行参数，并将之解释为XTA指令。例如，使用以下指令，运行二进制（test.xe）XTA脚本（script.xta）：

```
xta -load test.xe -source script.xta -exit
```

备注：字符“-”被用作指令之间的分隔符。

13.5.3 嵌入指令到源中

XTA可以使用指令编译器指令，嵌入指令到源代码中。例如，

```
#pragma xta command "print summary"
```

嵌入源中的所有指令均在二进制加载到XTA时运行。指令的执行次序为在文件中出现的次序，但是未定义不同来源文件中的指令之间的次序。

仅支持采用XC代码的编译器指令。

13.6 通过Jython接口编辑XTA脚本

XTA支持使用Jython语言（在Java虚拟机器上运行的Python的实施）撰写的脚本。XTA Jython脚本的扩展名必须为.py。它们的执行方式与机遇指令的XTA脚本的相同。从Jython，可以通过全局可触及xta对象，得到XTA特点。实例脚本参见图25。此脚本将二进制test.xe加载到XTA中，并分析函数function Name。然后，它设置每个产生的路由的回路计数，最后打印每个回路的最佳和最差事件执行时间。

```
import sys
import java

try :
    xta . load (" test .xe");

except java . lang . Exception , e:
    print e. getMessage ()

try :
    ids = xta . analyzeFunction (" functionName ");

    for id in ids :
        xta . setLoop (id , " loopReference ", 10)

    for id in ids :
        print xta . getRouteDescription (id),
        print xta . getWorstCase (id , "ns"),
        print xta . getBestCase (id , "ns")

except java . lang . Exception , e:
    print e. getMessage ()
```

图25：XTA Jython脚本的实例。

14 XTA 指令行手册

本章内容：
· 指令
· 编译器指令
· 计时模式
· 回路范围
· 引用类
· XTA Jython接口
· 代码引用语法

本章列出了XTA支持的所有指令和选项、引用类及语法引用。

14.1 命令

14.1.1 添加

add branch <from BRANCH> [<to INSTRUCTION>]+
添加给出的from/to引用到分支清单中。
add tile <tile id|*>
添加xCORE窗口并联到活动设置
add exclusion <ANY>
添加给定的引用到排斥列表
add functiontime <FUNCTION> <value> <MODE>
添加给定的函数执行时间到定义列表
add instructiontime <ENDPOINT> <value> <MODE>
添加给定的指令执行时间到定义列表
add loop <ANY> <iterations>
添加给定的回路计数定义到定义列表
add looppath <ANY> <iterations>
添加给定的回路路径计数定义到定义列表
add loopscope <ANY> <SCOPE>
添加给定的回路范围定义到定义列表
add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>
添加给定的路径执行时间到定义列表

14.1.2 分析

analyze endpoints <from ENDPOINT> <to ENDPOINT>
在规定端点之间分析
analyze function <FUNCTION>
分析给定的函数
analyze loop <ANY>

分析给定的回路

14.1.3 配置

config case <best/worst>
设置事件（当前：最差）

config Ewarning <on/off>
是否将错误处理为警告（当前：关闭）

config freq <node id> <tile frequency>
设置给定节点的操作频率，单位MHz

config from <ENDPOINT>
设置开始端点

config looppoint <ANY>
设置回路点

config scale <true/false>
配置结构是否按比例（当前：是）

config srcpaths <paths>
设置（分号隔离）源检索路径

config cores <tile id> <num cores>
设置给定窗口并联当前执行的核心数量

config timeout <seconds>
设置加载上的工具超时

config Terror <on/off>
是否将计时故障处理为错误（当前：打开）

config to <ENDPOINT>
设置结束端点

config verbosity <level>
设置工具信息级别（范围：-10 -> +10，默认：0）

config Werror <on/off>
是否将警告处理为错误（当前：关闭）

14.1.4 清除

clear（）
清除屏幕（仅GUI模式）

14.1.5 调试

debug dumpactiveexclusions()
导出已解决的排斥的PC列表

debug dumpcachedfunction <FUNCTION>
导出缓冲函数结构

debug dumpcallgraph()
导出所有窗口并联的调集图，采用点（图形）格式

debug dumpcontrolflow <FUNCTION>

导出给定函数的控制流图，采用点（图形）格式
debug dumpmanual()
导出手册的控制台引用章的，采用tex格式
debug dumpstacknodes <REFERENCE>
导出给定引用的堆栈节点
debug dumpunresolved instructions（）
导出一份未解决指令的列表
debug verifyreference <ANY>
认证存在给定引用
debug frompoints（）
显示当前配置的开始端点
debug topoints（）
显示当前配置的结束端点
debug instructiontime <route id> <node id>
显示给定路由中给定节点的指令执行时间设置
debug loop <route id> <node id>
显示给定路由中给定节点的回路迭代设置
debug looppath <route id> <node id>
显示给定路由中给定节点的回路路径迭代设置
debug loopscope <route id> <node id>
显示给定路由中给定节点的回路范围设置
debug listglobalreferences <ANY>
列出全局树给定引用的所有匹配引用
debug listroutereferences <route id> <ANY>
列出给定路由上给定引用的所有匹配引用
debug memusage()
显示HVM的当前内存使用
debug getmemthreshold()
显示当前内存使用阈值
debug setmemthreshold <threshold>
设置内存阈值到给定数值（0.0 - 1.0）

14.1.6 重复
echo "text"
打印文本到控制台

14.1.7 退出
exit（）
退出该引用

14.1.8 帮助

help [command|command subcommand|option]
显示给定参数的帮助消息

14.1.9 历史记录
history（）
显示指令历史

14.1.10 加载
load <xe file>
加载给定的XMOS可执行文件
14.1.11 列表
list allcalls（）
列出所有可能的调用位置
list allendpoints（）
列出所有可能的端点位置
list branches [route id]
列出规定路由可选择的分支
list calls（）
列出调用
list tiles（）
列出活动xCORE窗口并联
list endpoints（）
列出端点
list exclusions [route id]
显示规定路由可选择的排斥
list functions()
列出加载应用中的函数
list functiontimes()
显示 函数执行时间定义
list instructiontimes()
显示 指令执行时间定义
list knowns <route id>
显示给定路由的已知设置的列表
list labels（）
列出标签
list loops()
显示回路定义
list looppaths()
显示回路路径定义
list loopscopes()
显示回路范围定义

list pathtimes()
显示路径时间定义
list sources（）
列出源文件
list srccommands（）
显示嵌入加载可执行文件中的指令列表
list srcloops（）
显示嵌入加载可执行文件中的回路计数
list srclabels（）
列出源标签
list allsrclabels（）
列出源标签的所有可能的位置
list corestartpoints（）
列出逻辑核心起点
list corestoppoints（）
列出逻辑核心终点
list 未知事件 <路由 id>
显示给定路由的未知事件列表

14.1.12 打印
print summary()
显示路由概要（信息级别 -2|-1|0）
print structure <route id> [node id]
显示给定路由/节点的结构（信息级别0|1）
print asm <route id> [node id]
显示给定路由/节点的带注解的集合
print src <route id> [node id]
显示给定路由/节点的带注解的源文件
print trace <route id> [node id]
显示给定路由/节点的最差路径的指令跟踪
print routeinfo <route id>
显示给定路由的详细信息
print nodeinfo <route id> [node id]
显示给定路由中给定节点的详细信息
print warnings（）
打印所有计时警告
print distribution <route id> [node id]
显示给定路由/节点的计时分布

14.1.13 pwd

pwd（）
显示当前工作目录

14.1.14 删除
remove branch <from BRANCH|*> [<to INSTRUCTION|*>]+
从分支列表中删除给定的from/to引用
remove tile <tile id|*>
从活动设置中删除xCORE窗口并联
remove exclusion <ANY|*>
从排斥列表中删除给定的引用（或所有if '*'）
remove functiontime <FUNCTION|*>
从定义列表中删除给定的函数执行时间
remove instructiontime <ENDPOINT|*>
从定义列表中删除给定的指令执行时间
remove loop <ANY|*>
从定义列表中删除给定回路计数定义
remove looppath <ANY|*>
从定义列表中删除给定的回路路径计数定义
remove loopscope <ANY|*>
从定义列表中删除给定的回路范围定义
remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>
从定义列表中删除给定的路径执行时间
remove route <route id>
从当前分析中删除带有给定id的路由

14.1.15 编码器
scripter disable <ANY>
失活一项映射
scripter dump（）
删除代表当前状态的脚本 – 将活动编译器指令嵌入源中
scripter embed <filename>
嵌入脚本到指定的文件中 – 也嵌入活动编译器指令到源中
scripter enable <ANY>
激活一项映射
scripter listrefs（）
列出脚本中使用的所有引用
scripter rename <ANY> <TO_NAME>
给映射重新命名

14.1.16 设置
set exclusion <route id> <ANY>

设置给定引用的排斥
set functiontime <route id> <FUNCTION> <value> <MODE>
设置给定路由上给定函数的计时要求
set instructiontime <route id> <ENDPOINT> <value> <MODE>
设置给定pc中指令的执行时间
set loop <route id> <ANY> <iterations>
设置所识别的回路的迭代次数
set looppath <route id> <ANY> <iterations>
设置所识别的路径的迭代次数
set loopscope <route id> <ANY> <SCOPE>
设置引用回路的范围
set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>
设置给定路由上给定路径的计时要求
set required <route id> <value> <MODE>
设置给定路由的最大许可执行时间

14.1.17 源
source <file name> [args]
给定脚本文件的源

14.1.18 状态
status（）
显示当前状态
14.1.19 版本
version（）
显示版本信息

14.2 编译器指令
#pragma xta label "name"
提供可用于规定计时限制的标签。
#pragma xta endpoint "name"
规定一个端点。它可以在输入或输出声明前出现。
#pragma xta call "name"
定义（函数）调集点的标签。用于规定特定调集函数。例如，如果函数中含有一个回路，则此回路的迭代可以设置为不同数值，并根据此不同的数值，确定调集函数的点。
#pragma xta command "command"
允许XTA指令嵌入源代码中。每次将二进制加载到XTA时，所有指令运行。依照在文件中出现的先后次序执行命令，但是未定义不同源文件中指令之间的次序。
#pragma xta loop "integer"
将给定回路XTA迭代用于含有编译器指令的回路。

XMOS

14.3 计时模式
可用计时模式有：
ns（）
纳秒
us（）
微秒
ms（）
毫秒
MHz（）
兆赫兹
KHz（）
千赫兹
Hz（）
赫兹
cycles（）
核心循环计数为执行该次序，逻辑核心需要的调度槽的数量。核心循环和时间之间的关系为当前运行的核心数与xCORE窗口并联频率的函数。

14.4 回路范围
范围所支持的数值为：
relative/r（）
扩散到闭路路径的迭代数（默认）
absolute/a（）
绝对迭代次数

14.5 引用种类
14.5.1 函数
FunctionPc（）
原程序计数器的规定格式：0x*
Function（）
任何函数

14.5.2 分支
EndpointPC（）
原程序计数器规定格式：0x*
CallPc（）
调用规定格式： 0x*
CallFileLine（）
调用规定格式： ‘file name：line number’

Call（）
调用规定使用源代码级编译器指令机制
Label（）
针对可执行节定义的任何源代码或汇编级符号
CallLabel（）
针对可执行节定义的任何源代码或汇编级符号
14.5.3 指令
EndpointPC（）
原程序计数器规定格式： 0x*
FunctionPc（）
原程序计数器规定格式： 0x*
Function（）
任何函数
Label（）
针对可执行节定义的任何源代码或汇编级符号

14.5.4 端点
EndpointPC（）
原程序计数器规定格式： 0x*
EndpointFileLine（）
端点规定格式： ‘file name：line number’
Endpoint（）
端点规定使用源代码级编译器指令机制
CallPc（）
调用规定格式： 0x*
CallFileLine（）
调用规定格式： ‘file name：line number’
Call（）
调用规定使用使用源代码级编译器指令机制
Label（）
针对可执行节定义的任何源代码或汇编级符号
CallLabel（）
针对可执行节定义的任何源代码或汇编级符号

14.5.5 ANY
SrcLabelPc（）
原程序计数器规定格式： 0x*
EndpointPC（）
原程序计数器规定格式： 0x*

XMOS

EndpointFileLine（）
端点规定格式： ‘file name：line number’
Endpoint（）
端点规定使用源代码级编译器指令机制
CallPc（）
调用规定格式： 0x*
CallFileLine（）
调用规定格式： ‘file name：line number’
Call（）
调用规定使用源代码级编译器指令机制
SrcLabelFileLine（）
源代码标签规定格式： ‘file name：line number’
SrcLabel（）
源代码标签规定使用源代码级编译器指令机制
Label（）
针对可执行节定义的任何源代码或汇编级符号
CallLabel（）
针对可执行节定义的任何源代码或汇编级符号

14.5.6 FUNCTION_WITH_EVERYTHING
EverythingReference（）
与任何函数匹配： ‘*’
函数Pc（）
原程序计数器规定格式： 0x*
函数（）
任何函数

14.5.7 BRANCH_WITH_EVERYTHING
EverythingReference（）
与任何分支匹配： ‘*’
EndpointPC（）
原程序计数器规定格式： 0x*
CallPc（）
调用规定格式： 0x*
CallFileLine（）
调用规定格式： ‘file name：line number’
Call（）
调用规定使用源代码级编译器指令机制
Label（）
针对可执行节定义的任何源代码或汇编级符号

CallLabel（）
针对可执行节定义的任何源代码或汇编级符号

14.5.8 INSTRUCTION_WITH_EVERYTHING
EverythingReference（）
与任何指令匹配：‘*’
EndpointPC（）
原程序计数器规定格式：0x*
FunctionPc（）
原程序计数器规定格式：0x*
Function（）
任何函数
Label（）
针对可执行节定义的任何源代码或汇编级符号

14.5.9 ENDPOINT_WITH_EVERYTHING
EverythingReference（）
与任何端点匹配：‘*’
EndpointPC（）
原程序计数器规定格式：0x*
EndpointFileLine（）
端点规定格式：‘file name：line number’
Endpoint（）
端点规定使用源代码级编译器指令机制
CallPc（）
调用规定格式：0x*
CallFileLine（）
调用规定格式：‘file name：line number’
Call（）
调用规定使用源代码级编译器指令机制
Label（）
针对可执行节定义的任何源代码或汇编级符号
CallLabel（）
针对可执行节定义的任何源代码或汇编级符号
14.5.10 ANY_WITH_EVERYTHING
EverythingReference（）
与任何ANY匹配：‘*’
SrcLabelPc（）
原程序计数器规定格式：0x*
EndpointPC（）

XMOS

原程序计数器规定格式： 0x*
EndpointFileLine（）
端点规定格式： ‘file name：line number’
Endpoint（）
端点规定使用源代码级编译器指令机制
CallPc（）
调用规定格式： 0x*
CallFileLine（）
调用规定格式： ‘file name：line number’
Call（）
调用规定使用源代码级编译器指令机制
SrcLabelFileLine（）
源代码标签规定格式： ‘file name：line number’
SrcLabel（）
源代码标签规定使用源代码级编译器指令机制
Label（）
针对可执行节定义的任何源代码或汇编级符号
CallLabel（）
针对可执行节定义的任何源代码或汇编级符号
14.6 XTA Jython接口
Jython与全局xta目标的接口如下：

14.6.1 加载方法

```
void load(String fileName) throws Exception
```

14.6.2 路由创建/删除方法

```
List<Integer> analyzeFunction(String functionName) throws Exception
List<Integer> analyzeEndpoints(String fromRef, String toRef) throws Exception
List<Integer> analyzeLoop(String loopRef) throws Exception
void removeRoute(int routeId) throws Exception
```

14.6.3 添加/删除方法

```
void addTile(String tileReference) throws Exception
void removeTile(String tileReference) throws Exception
Collection<String> getTiles() throws Exception

void addExclusion(String ref) throws Exception
void removeExclusion(String ref) throws Exception
Collection<String> getExclusions() throws Exception

void addBranch(String fromRefString, Collection<String> toRefStrings)
throws Exception
void removeBranch(String fromRefString, Collection<String> toRefStrings)
throws Exception
Collection<String> getBranches() throws Exception
Collection<String> getBranchTargets(String branch) throws Exception

void addLoop(String ref, long iterations) throws Exception
void removeLoop(String ref) throws Exception
Collection<String> getLoops() throws Exception

void addLoopPath(String ref, long iterations) throws Exception
void removeLoopPath(String ref) throws Exception
Collection<String> getLoopPaths() throws Exception

void addLoopScope(String ref, boolean absolute) throws Exception
void removeLoopScope(String ref) throws Exception
Collection<String> getLoopScopes() throws Exception

void addInstructionTime(String ref, double value, String units)
throws Exception
void removeInstructionTime(String ref) throws Exception
Collection<String> getInstructionTimes() throws Exception
```

```
void addFunctionTime(String ref, double value, String units)
throws Exception
void removeFunctionTime(String ref) throws Exception
Collection<String> getFunctionTimes() throws Exception

void addPathTime(String fromRef, String toRef, double value, String units)
throws Exception
void removePathTime(String fromRef, String toRef) throws Exception
Collection<String> getPathTimes() throws Exception
```

14.6.4 设置方法

```
void setRequired(int routeId, double value, String units) throws Exception
void setFunctionTime(int routeId, String refString, double value,
String units) throws Exception
void setPathTime(int routeId, String fromRef, String toRef, double value,
String units) throws Exception
void setInstructionTime(int routeId, String refString, double value,
String units) throws Exception
void setLoop(int routeId, String refString, long iterations)
throws Exception
void setLoopPath(int routeId, String refString, long iterations)
throws Exception
void setLoopScope(int routeId, String refString, boolean absolute)
throws Exception
void setExclusion(int routeId, String refString) throws Exception
```

14.6.5 获得方法

```
double getRequired(int routeId, String units) throws Exception
double getWorstCase(int routeId, String units) throws Exception
double getBestCase(int routeId, String units) throws Exception
List<String> getWarnings(int routeId) throws Exception
List<String> getErrors(int routeId) throws Exception
List<Integer> getRouteIds() String getRouteDescription(int routeId)
throws Exception
```

14.6.6 配置方法

```
void configCores(String tileReference, int numCores) throws Exception
void configFreq(String nodeId, double tileFrequency) throws Exception
```

14.7 代码引用语法

代码引用由回溯和基础引用格式组成：

code-ref	::=	*back-trail base-ref*
back-trail	::=	*base-ref*
	|	*base-ref* , *back-trail*
base-ref	::=	*pc-ref*
	|	*label-ref*
	|	*function-ref*
	|	*endpoint-ref*
	|	*srclabel-ref*
	|	*call-ref*
pc-ref	::=	*pc-class hex-constant*
label-ref	::=	*label-class label-string*
function-ref	::=	*function-class function-name*
	|	*function-class hex-constant*
endpoint-ref	::=	*endpoint-class file-line*
	|	*endpoint-class endpoint-label*
	|	*endpoint-class hex-constant*
srclabel-ref	::=	*srclabel-class label-string*
call-ref	::=	*call-class label-string*
	|	*call-class hex-constant*
pc-class	::=	
	|	*PC:*
label-class	::=	
	|	*LABEL:*
functionclass	::=	
	|	*FUNCTION:*
endpointclass	::=	
	|	*ENDPOINT:*
srclabelclass	::=	
	|	*SRCLABEL:*
call-class	::=	
	|	*CALL:*
file-line	::=	*file-name : integer-constant*

F部分
硬件上运行

内容

15 使用 xTIMEcomposer 运行程序

本章内容
创建一个运行配置
重新运行一个程序

xTIMEcomposer 使用 Run Configurations 确定用于运行程序的设置。运行配置是项目和目的平台特有的。

15.1 创建一个运行配置

使用以下步骤创建一个运行配置：

1.在项目 Explorer 中选择一个项目

2. 选择 Run——Run Configurations。

3. 在左边的面板中双击 XCore Application。

xTIMEcomposer 创建一个新的配置并在右边的面板中显示默认设置 ，如图 10 所示。

4. 在 Name 中输入配置的名称。

5. xTIMEcomposer 尝试为你识别目的项目和可执行项目。单击项目文本框右边的 Browse 并在项目 Selection 对话框中选择需要的项目。然后，单击 Search Project，并在 Program Selection 对话框中选择可执行文件。

⚠你必须之前已经编译了你的程序，且可选择的可执行程序无任何错误。

6. 如果你有一个连接到你的系统的开发板，则应检查 hardware 选项并从 Target 清单中选择你的调试适配器。或者，检查 simulator 选项以在 XMOS 模拟器上运行你的程序。

7. 单击 Run。

xTIMEcomposer 加载你的可执行程序，并在 Console 中显示程序生成的任何输出。

XMOS

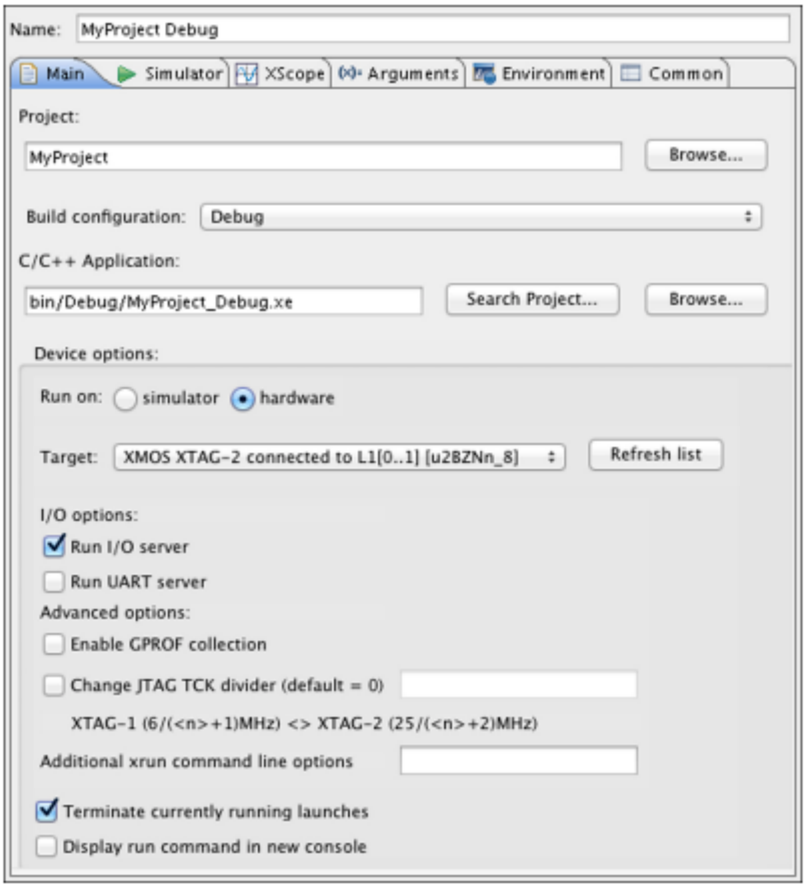


图 26：运行配置窗口

15.2 重新运行一个程序

xTIMEcomposer 记住上次运行程序时使用的配置。想要再次使用相同的配置，只需单击 Run 按钮。想要使用不同的配置，单击 Run 按钮右边的箭头并从下拉清单中选择一种配置。

16 XRUN 命令行手册

本章内容

XRUN 加载 XMOS Executable（XE）文件到目的硬件并在上面运行。它需要安装 XMOS 或 FTDI USB-to-JTAG 驱动器，具体视随目的硬件使用的适配器而定（参见§2）。

16.1 总体选项

以下选项被用于规定将要运行的可执行选项，且可以选择运行程序的 xCORE 展开图。

xe-file 规定加载和运行的 XE 文件。

--verbose 打印有关加载到目的设备的程序的信息。

--help 打印所支持的命令行选项的说明。

--version 显示版本号和版权。

16.2 目的选项

以下选项用于规定目的硬件平台。

--list-devices
-l 打印连接到主机的所有 JTAG 适配器和每个 JTAG 链上的设备的编号清单，形式为：

标识	名称	适配器标识	设备
--	----	----------	-------

适配器依照其序列号排序。

--list-board-info
-lb 显示有关所连接的目的板的信息。

--id ID 规定连接到目的硬件上的适配器。

--adapter-id ADAPTER-SERIAL-NUMBER
规定连接到目的硬件上的适配器的序列号。

--jtag-speed n
将 JTAG 时钟的驱动器设计为 n。如有规定，默认数值为 0，最大数值为 70。
如果是基于 FTDI 的调试适配器，则 JTAG 时钟速度设置为 6/(n+1)MHz。如果是基于 XMOS 的调试适配器，则 JTAG 时钟速度设置为 25/（n+1）MHz。

--noreset 在加载程序前，切勿在 JTAG 扫描链上重置 XMOS 设备。这不是默认的。

16.3 调试选项

以下选项可用于激活调试能力。

--io 使 XRUN 在加载到程序后保持连接到 JTAG 适配器上，并激活主机的系统调集。当程序调集退出后 XRUN 终止。
默认在加载程序后，XRUN 与 TAG 适配器断开。

--uart 激活与 XMOS USB-to-JTAG 适配器上的 UART-to-USB 转换器连接的 UART 服务器。转换器的操作速率为 115200 bits/sec。
XMOS 适配器上的 USB-to-UART 转换器在 XSYS 连接器上有两个插脚（在 XMOS 开发板上）连接到 XMOS 设备的端口上。端口在 XN 文件中的命名为 PORT_UART_TX 和 PORT_UART_RX。
基于 FTDI 芯片的适配器不支持此选项。

--attach 连接到 JTAG 适配器（运行程序），使用主机激活系统调集。当程序执行调集退出时，XRUN 终止。
必须使用此选项规定 XE 文件。

--dump-state
在 JTAG 扫描链中打印所有 xCORE 展开图的核心、寄存器和堆栈内容。

16.4 XScope 选项

以下选项可用于激活 XScope 能力。

--xscope 激活一个带有目的的 XScope 服务器。

--xscope-realtime
使用插座连接，激活带有目的的 XScope 服务器

--xscope-file filename
规定了用于 XScope 数据收集的文件名。

--xscope-port ip：port
规定了用于实时数据采集的 IP 地址和端口。

--xscope-limit limit
规定了 XScope 数据收集的记录极限。

第 G 部分
应用仪表和微调

内容

17 使用 xTIMEcomposer 和 XScope 实时追踪数据

本章内容

xTIMEcomposer 和 XScope 库使你可以为程序装备实时收集应用数据的探针。此数据可以通过 XTAG-2 调试适配器发送到 xTIMEcomposer 以实时显示或写入文件以进行离线分析。

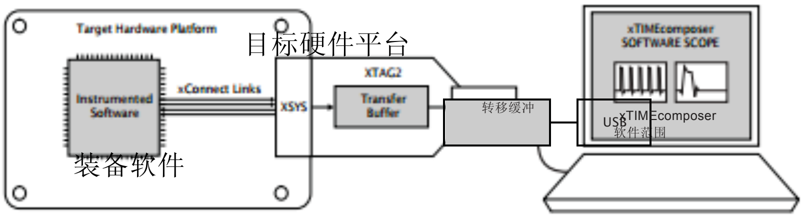


图 27：xScope 连接性

如果您正在使用传统FTDI或XTAG-1调试适配器或您的目的硬件上的XSYS连接器未提供xConnect链接，则您可以配置探针，使之在您的适配器的UART界面输出追踪数据（参见XM-000957-PC）。需要说明的是，仅单展开图支持UART界面，并大大降低性能。

17.1 XN文件配置

为了使工具配置使用xscope进行高速数据收集需要的xConnect链接，板的XN文件必须修订以暴露出与XTAG-2设备的连接。必须将以下信息添加到板的XN文件链接部分以设置目的设备使用的连接，以与XTAG-2和xscope信道通信。

```
<Link Encoding■"2wire" Delays■"4,4" Flags■"XSCOPE">
  <LinkEndpoint NodeId■"0" Link■"XOLD"/>
  <LinkEndpoint RoutingId■"0x8000" Chanend■"1"/>
```

```
</Link>
```

需要说明的是，当链接设置为2线时，最短延迟设置为4且标记规定了此链接将用于流水化调试。如果延迟数值设置的太高，将导致xscope使用的包的输出频率更低。RoutingId也很重要，因为数值0x8000规定了这是专门用于xscope的特殊链接。

当用于多展开图系统时，必须规定连接到XSYS连接器的包的NodeId。工具使用其他展开图设置链接，但是它们需要知道那个设备有与XTAG-2的外部链接。

17.2 给程序装备仪表

图28中的实例程序使用XScope仪表函数追踪麦克风的输入水平。

图 28： 追踪麦克风的输入水平的程序

```
#include <xscope.h>

port micL;
port micR;

void xscope_user_init(void) {
  xscope_register(2,
    XSCOPE_CONTINUOUS, "Microphone Left",  XSCOPE_UINT, "mV",
    XSCOPE_CONTINUOUS, "Microphone Right", XSCOPE_UINT, "mV"
  );
}

int main() {
  while (1) {
    int sample;
    micL :> sample;
    xscope_probe_data(0, sample);
    micR :> sample;
    xscope_probe_data(1, sample);
  }
}
```

构造器 xscope_user_init 寄存两个探针以追踪麦克风左边和右边的输入。探针定义为 **continuous**，也即 xTIMEcomposer 可以在两次连续测量结果之间插入数值。这些探针定义为采集典型 unsigned int 数值。

大体上来讲，程序每次调集探针函数 xscope_probe_data 时，就会从麦克风采集一次数据。此函数创建了一份追踪记录并将之发送到 PC。

XMOS

图 29 总结了可以配置的各种类型的探针。仅 **continous 探针可以实时显示。**

图 29: 所支持的探针类型

探针类型	**数据类型**	**Scope 视图**	**实例**
XSCOPE_CONTINUOUS	XSCOPE_UINT XSCOPE_INT XSCOPE_FLOAT	线图。可以插入。	电机控制器的电压水平
XSCOPE_DISCRETE	XSCOPE_INT	水平线	音频CODEC的缓冲水平
XSCOPE_STATEMACHINE	XSCOPE_UINT	状态机器	协议的进度
XSCOPE_STARTSTOP	XSCOPE_NONE XSCOPE_UINT XSCOPE_INT XSCOPE_FLOAT	启动/停止栏	记录功能输入和退出，带有可选标签数值

17.3 启用追踪后的程序配置与运行

在装备您的程序的时候，您必须汇编并将之与XScope库链接，并离线或实时运行它。

依照下面的步骤链接XScope library和run XScope：

1. 打开您的项目的Makefile 。
2. 找到您构建配置的XCC_MAP_FLAGS_*config*变量，例如 XCC_MAP_FLAGS_Release。
3. 添加选项 –fxscope。
4. 为您的目的设备创建一个运行配置（参见§15.1）。
5. 单击**XScope**标签并选择**Offline Mode**以保存数据到文件以进行离线分析，或**Real-Time Mode**以输出数据到实时查看器。

   在离线模式下，xTIMEcomposer记录追踪数据直到程序终止并保存追踪数据到文件xscope.xmt。想要变更时，在**Output file**文本框中输入文件名。在**Limit records to**文本框中输入数字以限制追踪文件的大小。

   在实时模式下，xTIMEcomposer打开Scope视图并在执行程序时，显示追踪数据的动画视图。
6. 单击**Run**以保存并运行配置。

XMOS

15.4 离线分析数据

双击项目*Explorer*中的追踪文件以在 **Scope视图**中打开，如图30所示。

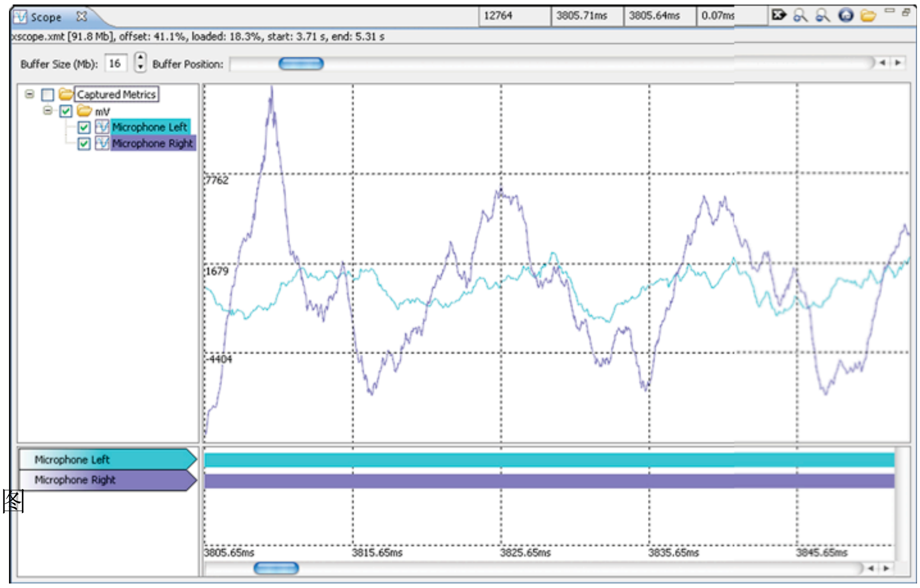


图30：
离线Scope视图

Scope视图的顶部面板显示每个选中的探针的数值的图形：*x* –轴代表时间（依照底部面板的时限）和*y* –轴代表追踪到的数值。探针依照赋予其的单位分组，且多个单位相同的探针可以覆盖到同一个图上。

移动指针到范围数据上，显示窗口顶部四个数字框中前两个的当前数据（*y* –值）和时间（*x* –值）。单击视图以显示红线标记 – 相关的时间显示在第三个框中。第三个数字框显示标记时间与当前指针位置之间的差异。

如果指针变为手指标签，双击以确定负责生成追踪点的源代码中的叙述。

此视图的底部面板显示每个探针的时限：探针时限上的垂直线表示探针创建记录的时间。

向左或右拖动**Buffer Position**滑块，穿过时限。升高**Buffer Size**框中的数值以在窗口中显示更多信息。

使用窗口顶部的**Scope视图**工具栏以执行额外任务：单击**Continuous points 按钮**，显示用于插入连续信号的数据点。

单击**Zoom Fit按钮**查看所有数据点。

单击**Open按钮**并浏览文件以加载并非您的项目的一部分的追踪文件。

17.5 实时分析数据

*Scope视图*可以实时显示从硬件流入的追踪数据。左边的面板显示信号信息和控制器，右边的面板显示信号的屏幕视图。

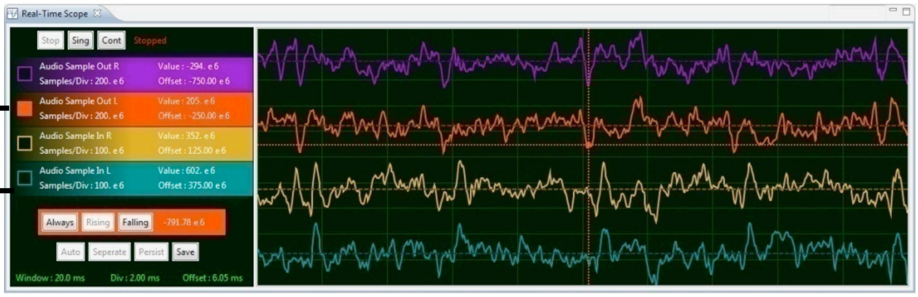


图 31：
实时 Scope 视图

左边的面板显示了应用寄存的*continous*探针的清单（参见§17.2）。每个命名的探针被赋予一种颜色，以在显示器上绘制事件并用于在屏幕面板中识别探针。

*Scope视图*基于传统示波器，并在触发器周围捕集数据和显示。捕集模式、显示模式、触发器和时间基准均在左边面板中控制。右边的面板中有10个水平和垂直部分，比例使用每个部分的单元数表示。

数值控制均可以使用鼠标修改：单击左边的按钮以升高数值，或单击右边的按钮以降低数值。如果您的平台太支持，则可以使用滚轮（Mac OS/X、Linux及部分而非全部Windows版本）。

17.5.1 捕集控制

有三种捕集模式：*continous*、*single capture或stopped*。默认的启动模式为系统连续捕集和显示。与捕集控制有关的标签显示XScope系统的当前状态。

图 32：
捕集控制器

停止显示

停止屏幕面板触发和捕集，设置此模式时无法继续更新屏幕。此模式可用于检查所捕集的数据。鼠标可用于变更新信号、时间基准比例和偏移（如下所述）以详细检查信号。停止后，您可以在时间基准上缩放并更加详细的查看信号：当时间基准较大，在时间基准上缩放将展示所有数据。

单次捕集
选择单次模式捕集一次屏幕数据就返回到停止状态。如果激活了触发器（参见图18），系统将等待满足此触发器条件，然后再更新屏幕并返回到停止状态。

连续捕集
选择自由运行模式以尽可能频繁的更新屏幕。如果触发器激活，则屏幕仅在满足触发器条件时更新。

17.5.2 信号控制

左边面板中显示的彩色标签上的每个注册探针可用的信号控制（参见图17）

图 33：
信号控制

激活/ 失活信号

双击名称切换新号的可见度。

信号样品/部分

使用鼠标按钮，更改此探针每个部分的样品数量，这影响信号的垂直比例。

信号屏幕偏移

使用鼠标按钮变更此探针的垂直偏移，这影响信号的垂直位置。

信号触发器

单击探针标签左边的触发器箱，可将信号用作触发器（参见图18）。只有一个信号可用于触发。

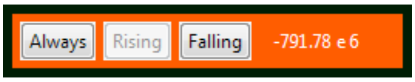
17.5.3 触发控制

触发器可用于限制系统，从而使仅在满足条件时捕集数据。默认所有触发器均失活，导致可以无条件的捕集数据。为了激活触发，必须单击探针标签左边的框以选择触发器。

当触发激活时，屏幕上出现十字线以显示触发器水平（相对于所选择的触发器的信号）和触发器在时间基准上的偏移。十字线的中心为触发器发生（现在或之前）的时间和数值；其左边为触发器之前的信号，而右边为触发器后面的信号。

可以直接单击右边的窗格直接设置触发器水平和偏移。变更仅在范围未停止时生效，而不论是连续运行还是设置为单一触发。

图 34：触发控制器



始终
失活触发器并无条件的捕集数据。

升高
触发信号的升高边缘。当选择将要用于触发的信号时，这是默认模式。

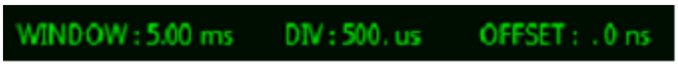
下降
在信号的下降边缘触发。

与激活触发器有关的数值标签显示信号的当前数值设置。这可以使用鼠标按钮变更。

15.5.4 时间基准控制

时间基准控制用于设置信号捕集窗口的时间范围，使您能够缩放和移动水平轴。

图35：
时间基准控制器



时间窗口

时间窗口的当前尺寸。缩放时间中的所有信号并影响每个部分的时间。

每个部分的时间

每个部分的时间单元。缩放时间中的所有信号并影响时间窗口。

时间窗口偏移

触发器在时间窗口中的位置。向左和向后转移所有信号。需要说明的是，触发器不可见，且应在时间窗口的左边或右边。仅在有限的数值下，信号向右移动。

17.5.5 屏幕控制

有几个命令可用于操作所有信号。

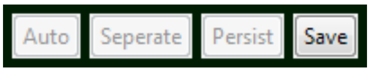


图 36:
屏幕控制器

自动范围信号

自从布置所有当前信号以与屏幕匹配。短时测量信号，每个信号缩放并偏移以与屏幕匹配。显示的所有信号均交叉。

单独的信号

与Auto Range类似，但是所有信号均缩放以与屏幕的一小部分匹配。所有信号均偏移，因此可以单独查看它们。

持久显示器

失活。

保存数据

保存当前scope视图到用户定义位置的PNG文件中。

17.6 使用UART接口追踪

如果您正在使用传统的FTDI或XTAG-1调试适配器，或如果您的目的硬件上的XSYS连接器未提供xConnect链接，则您可以通过您的适配器提供的UART接口输出数据。

想要使用UART接口，您必须向XScope库提供一个使用连接到您的调试适配器上的UART-TX插脚初始化的1比特的UART TX端口。初始化的实例如下。

```
#include <platform.h>
#include <xscope.h>

port uart_tx = PORT_UART_TX;

void xscope_user_init(void) {
  xscope_register(2,
    XSCOPE_CONTINUOUS, "Microphone Left",  XSCOPE_UINT, "mV",
    XSCOPE_CONTINUOUS, "Microphone Right", XSCOPE_UINT, "mV"
  );
  xscope_config_uart(uart_tx);
}
```

因为UART接口使用端口，而不是xConnect链接，因此只需一个扩展图就可以调集探针函数。

18 XScope 性能数据

本章内容

从xCORE设备转移到调试适配器的数据无损失，但是从调试适配器转移到您的主机PC的数据可能会出现损失，具体视您的PC的速度而定。

18.1 xCORE数据块和XTAG-2之间的转移速率

所建议的大部分目的硬件的过渡之间的xConnect链接速度为10ns（10MByte/sec）。这可以通过设置链接比特间间隙到5个周期实现（参见§46.5）。此速度下使用xConnect链接的探针函数的传统和最大调集速度见图37。

探针函数	等待时间(核心循环） 最大调集次数/秒		
xscope_probe_data_pred	15	（始终）	666,000
xscope_probe	20	（无争用）	999,000
xscope_probe_cpu	27	（无争用）	666,000
xscope_probe_data	22	（无争用）	666,000
xscope_probe_cpu_data	28	（无争用）	555,000

图 37：带有5循环比特间间隙的xConnect 链接的XScope 性能数据

如果连续调集两次，则第二次调集可能依照最大频率延迟。例如，如果调集xscope_probe_data_pred两次，第二次调集延迟约1.5µs。

可以通过加速链接并缩小比特间间隙以增加最大调集速度（参见§46.5）。小比特间间隙需要认真布置链接，因为它会提高链接频率。

UART接口执行速率为2MB/s。

18.2 XTAG-2和Host PC之间的转移速率

很多个人计算机仅限于以500,000项追踪记录/秒或更低的速度从XTAG-2输入追踪数据。如果您的个人计算机无法保持此速率，则可能丢失记录，降低追踪该数据的间隔尺寸。XDE Scope视图在时间线上标记数据损失。

XMOS

19 Xscope库API

本章内容

函数

列举

19.1 函数

void xscope_bytes(unsigned char id,
unsigned int size,
const unsigned char data[])

为含字节数组且具有逻辑核心的XSCOPE探针发送追踪事件。

此函数具有以下参数：

标识	XSCOPE探针标识
尺寸	用户数据尺寸
数据	用户数据字节（char[]）

Void xscope_char(unsigned char id, unsigned char data)

为char类型的特定XSCOPE探针发送追踪事件。

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（char）

void xscope_config_io (unsigned int mode)

配置XScope输入/输出重新定向。

此函数具有以下参数：

模式	输入/输出重新定向模式

void xscope_core_bytes(unsigned char id,
unsigned int size,
const unsigned char data[])

为char类型的特定XSCOPE探针发送追踪事件。

此函数具有以下参数：

标识	XSCOPE探针标识
尺寸	用户数据尺寸
数据	用户数据值（char）

void xscope_core_char(unsigned char id, unsigned char data)

为char类型且具有逻辑核心的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识	XSCOPE探针标识
尺寸	用户数据尺寸
数据	用户数据值（char）

void xscope_core_double(unsigned char id, double data)

为double类型且具有逻辑核心的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（double）

void xscope_core_float(unsigned char id, float data)

为float类型且具有逻辑核心的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（float）

void xscope_core_int(unsigned char id, unsigned int data)

为int类型且具有逻辑核心的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（int）

void xscope_core_longlong(unsigned char id, unsigned long long data)

为long long类型且具有逻辑核心的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（long long）

void xscope_core_short(unsigned char id, unsigned short data)

为short类型且具有逻辑核心的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（long long）

void xscope_core_start(unsigned char id)

为具有逻辑核心的XSCOPE探针启动追踪模块

此函数具有以下参数：

标识	XSCOPE探针标识

void xscope_core_start_int(unsigned char id, unsigned int data)

为具有逻辑核心且类型为int的XSCOPE探针启动追踪模块

此函数具有以下参数：

标识	XSCOPE探针标识
数据	用户数据值（int）

void xscope_core_stop(unsigned char id)

为具有逻辑核心的XSCOPE探针停止追踪模块

此函数具有以下参数：

标识 XSCOPE探针标识

void xscope_core_stop_int(unsigned char id, unsigned int data)

为具有逻辑核心且类型为int的XSCOPE探针停止追踪模块

此函数具有以下参数：

标识 XSCOPE探针标识

数据 用户数据值（int）

void xscope_disable()

禁用本地xCORE数据块上的XSCOPE数据捕获

void xscope_double(unsigned char id, double data)

为double类型的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识 XSCOPE探针标识

数据 用户数据值（double）

void xscope_enable()

启动本地xCORE数据块上的XSCOPE事件捕获

void xscope_float(unsigned char id, float data)

为float类型的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识 XSCOPE探针标识

数据 用户数据值（float）

void xscope_int(unsigned char id, unsigned int data)

为int类型的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识 XSCOPE探针标识

数据 用户数据值（int）

void xscope_longlong(unsigned char id, unsigned long long data)

为long long类型的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识 XSCOPE探针标识

数据 用户数据值（long long）

void xscope_ping()

生成XSCOPE网络延迟系统时间戳事件

void xscope_register(int num_probes, ...)

向主机系统注册追踪探针

首个参数为需注册的探针个数。其他参数以四个为一组。

示例：

XMOS

```
xscope_register (1, XSCOPE_DISCRETE, "A probe", XSCOPE_UINT, "value");

xscope_register (2, XSCOPE_CONTINUOUS, "Probe", XSCOPE_FLOAT, "Level",
                 XSCOPE_STATEMACHINE, "State machine", XSCOPE_NONE, "
                 no name");
```

此函数具有以下参数：

num_probes　　规定的探针数量

void xscope_short(unsigned char id, unsigned short data)

为short类型的XSCOPE探针发送追踪事件

此函数具有以下参数：

标识　　XSCOPE探针标识

数据　　用户数据值（short）

void xscope_start(unsigned char id)

为特定XSCOPE探针启动追踪模块

此函数具有以下参数：

标识　　XSCOPE探针标识

void xscope_start_int(unsigned char id, unsigned int data)

为类型为int的XSCOPE探针启动追踪模块

此函数具有以下参数：

标识　　XSCOPE探针标识

数据　　用户数据值（int）

void xscope_stop(unsigned char id)

为XSCOPE探针停止追踪模块

此函数具有以下参数：

标识　　XSCOPE探针标识

数据　　用户数据值（int）

void xscope_stop_int(unsigned char id, unsigned int data)

为类型为int的XSCOPE探针停止追踪模块

此函数具有以下参数：

标识　　XSCOPE探针标识

数据　　用户数据值（int）

19.2 列举

xscope_IORedirectionMode

列举所有输入/输出重新定向模式。

此类型具有以下数值：

XSCOPE_IO_NONE

输入/输出未重新定向。

XMOS

XSCOPE_IO_BASIC 基础输入/输出重新定向。

XSCOPE_IO_TIMED 计时输入/输出重新定向。

XSCOPE_IO_NONE 输入/输出未重新定向

XSCOPE_IO_BASIC . 基础输入/输出未重新定向

XSCOPE_IO_TIMED 计时输入/输出未重新定向。

xscope_UserDataType
列举所有用户数据类型。

此类具有以下数值：

XSCOPE_NONE 无用户数据。

XSCOPE_UINT 无符号 int用户数据。

XSCOPE_INT 有符号的int 用户数据。
XSCOPE_FLOAT 浮动点用户数据。
XSCOPE_NONE 无用户数据
XSCOPE_UINT 无符号 int用户数据
XSCOPE_INT 有符号的int 用户数据。
XSCOPE_FLOAT 浮动点用户数据

xscope_EventType

列举所有类型的xscope事件。这种类型具有以下数值：

XSCOPE_STARTSTOP
启动/停止 - 事件获取一个起停值，代表执行模块

XSCOPE_CONTINUOUS
连续 - 仅启动一个事件，单时间戳“ping”。

XMOS

XSCOPE_DISCRETE
离散 – 自上一个事件起，事件生成一个离散模块。

XSCOPE_STATEMACHINE
状态机器 - 为每个新数值创建一个新事件状态。

XSCOPE_HISTOGRAM
XSCOPE_STARTSTOP
启动/停止 - 事件获取一个起停值，代表执行模块

XSCOPE_CONTINUOUS
连续 - 仅启动一个事件，单时间戳“ping”

XSCOPE_DISCRETE
离散 – 自上一个事件起，事件生成一个离散模块。

XSCOPE_STATEMACHINE
状态机器 - 为每个新数值创建一个新事件状态。

XSCOPE_HISTOGRAM

第H部分
模拟

内容

20 使用xTIMEcomposer模拟程序

本章内容

xCORE模拟器提供了使用一个或多个xCORE设备构建的接近准确循环的系统模型。您可以使用此模拟器查看处理器指令追踪、可视化系统状态及配置回路以模拟连接到XMOS端口和链接的组件的行为。

20.1 模拟器配置

使用下面的步骤配置模拟器：

1. 在项目**Explorer**中选择一个项目。
2. 选择**Run** ▶ **Run Configurations**。
3. 在左边的面板双击**XCore Application**。xTIMEcomposer创建一个新配置并在右边的面板显示默认设置。
4. 在右边的面板的**Name**中输入配置的名称。
5. xTIMEcomposer尝试尝试为您识别目的项目和可执行项目。单击项目文本框右边的**Browse**并在项目**Selection对话框**中选择需要的项目，然后单击**Search Project**并在**Program Selection对话框**中选择可执行的文件。您必须之前已经汇编了您的程序，且可选择的可执行文件无任何错误。
6. 选择**simulator**选项并单击**Simulator**标签以配置额外选项，如图22所示。

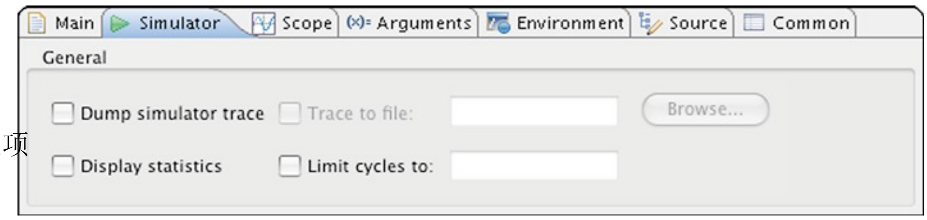


图 38：模拟器配置选项

选择**Dump simulator trace**以在模拟期间输出处理器指令追踪。
默认为在**Console**.中显示指令追踪。如果想要将追踪写入文件中，则应选择**Trace to file**并输入文件名。该文件名必须与您的项目中的所有其他文件都不同。
指令追踪的格式如图26所示。
想要在程序终止时查看程序执行总结时，选择**Display statistics**。该总结包括每个逻辑核心的指令计数及通过开关发送的数据和进行的控制的次数。
为了限制模拟器所执行的循环的次数，则应在文本框**Limit cycles to**中输入一个数值。如果您想程序从始到终运行，则应留白。这对无线循环中的模拟程序有用。

7. 想要保存和运行配置，单击**Run**。

xTIMEcomposer加载您的可执行文件，并在**Console**中显示您的程序所生成的任何输出。

xTIMEcomposer记住上次运行程序时使用的配置。想要再次使用相同的配置，只需单击**Run**按钮。想要使用不同的配置，单击**Run**按钮右边的箭头并从下拉清单中选择一种配置。

20.2 信号追踪

模拟器能够输出信号追踪数据到VCD文件，以便您使用xTIMEcomposer波形查看器可视化。

20.2.1 启用信号追踪

遵循下面的步骤激活模拟期间的信号追踪：

1. 创建一个模拟器**Run Configuration** （参见§20.1）。

2. 在**Signal Tracing**面板的**Simulator**标签中选择**Enable signal tracing**。
   想要追踪所有输入/输出插脚，在**System Trace选项**组中选择**Pins**。
   想要追踪特殊核心的机器状态，在**Core Trace选项**组中单击**Add**以显示一组可配置下拉菜单和复选框。然后，选择您想追踪的核心和机器。您可以追踪过程循环、端口、核心、时钟模块、垫和处理器指令。

3.单击**Run**。

xTIMEcomposer加载您的程序到模拟器，并在终止时，添加所生成的VCD文件到您的项目。

20.2.2 查看追踪文件

在项目**Explorer**中双击VCD文件以将之在**Signals视图**中打开，如图39所示。

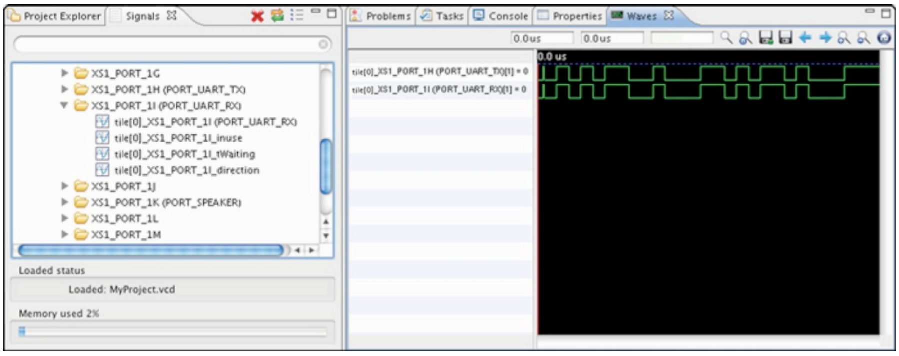


图 39：
信号和波形视图

在**Signals视图**中单击**plus sign** （Windows）或**disclosure triangle** （Mac）以扩展文件夹并显示其信号或子文件夹集合。双击信号或整个文件夹以在**Waves视图**中显示。

 单击**Display**按钮以在层次和平面视图之间切换。

20.2.3 查看信号

在**Waves视图**中移动指针到信号上以在Waves视图顶部的右边数值控制中开始模拟。如果指针编程一个手指标签，则应可以双击以确定负责驱动信号的源代码中的输出叙述。使用**Waves视图**工具栏进行以下操作：

 单击**Zoom Fit按钮**查看整个波形图。

 单击**Next和Previous按钮**以在选中的信号之间过渡。负责过渡的输出叙述在编辑器中高亮显示。

 单击**Search Transition按钮**打开一个对话框以检索特殊过渡，输入一个数值并单击**Find**。

 单击**Write Session File按钮**并输入该文件的文件名以保存配置。在加载VCD文件到**Waves视图**中时，保存您的设置以便使用。单击**Read Session File按钮**以加载最近保存的设置文件。

您可以通过以下方式控制信号在**Waves视图**中的显示方式：

使用ASCII显示信号数值：右击**Waves视图**中的信号以弹出右键菜单并选择**Data Format ▸ ASCII**。

在信号之间添加分隔符： 右击Waves视图上的信号以弹出菜单并从中选择 **Add Separator**。

给分隔符命名：右击分隔符以弹出菜单并选择**Name Separator**。在**Name Separator对话框**中输入分隔符的名称并单击**OK**。

移动分隔符：单击并拖动分隔符到需要的位置。

20.3 设置回路

您可以在模拟中连接任何两个端口或插脚，以模拟插脚之间的连接。依照以下步骤配置回路：

1. 创建模拟器**Run Configuration** （参见§20.1）。
2. 单击**Simulator**标签以显示模拟器配置选项。
3. 单击**Plugins**面板中的**Loopback**标签并选择**Enable pin connections**。
4. 在**Pin Connections**面板中单击**Add**。显示一个空的回路配置。回路由两套您可以用于两个不同的端口的选项组成。

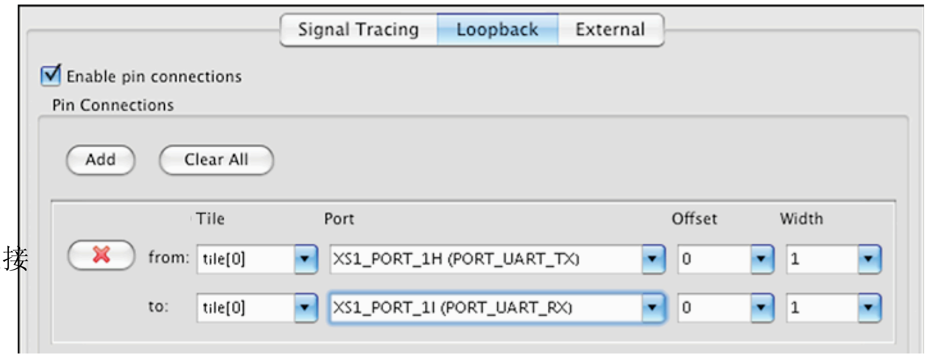


图 40：
设置回路连接

XMOS

5. 在每个连接末端的下拉清单中选择**Tile**和**Port**的数值。如果您不规定展开图，则从项目的XN文件选择端口清单，并自动确定展开图。如果您规定了展开图，则从标题文件中获取端口清单<xs1.h>。想要规定仅与端口连接的插脚的子集时，变更**Offset** 和**Width**的数值。

6. 单击**Run**。

20.4 配置模拟器插件

您可以连接模拟器到任何已经使用XMOS模拟器插入接口汇编到您的主机PC上的任何外部插入的模拟器。依照如下步骤配置外部插入：

1. 创建一个模拟器**Run Configuration** （参见§20.1）。

2. 单击**Simulator**标签以显示模拟器配置选项。

3. 在**Plugins**面板中单击**External**标签。

4. 单击**Add**以打开插入型配置对话框。

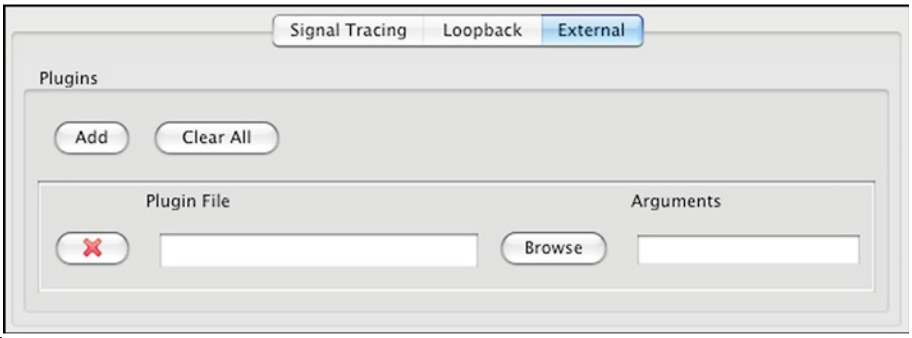


图 41:
设置外部插入

5. 选择插入DLL并规定可选命令行参数串。

6. 单击**Run**以保存您的设置，并使用规定的插入在模拟器上运行您的程序。

21 XSIM命令行手册

本章内容

XSIM执行了XMOS Executable （XE）文件的基于周期的模拟。XE文件中含有对目的硬件的说明。

21.1 总体选项

xe-file 规定将要模拟的XE文件。

--max-cycles *n*

当达到*n*系统周期时退出。

--plugin *name* *args*

加载一个插入DLL。*Args*的格式由插入确定；如果*args*中含有任何空格，则必须将之附在引述中。

--stats 退出时打印以下内容：

- 每个逻辑核心的指令计数的明细表。
- 通过开关发送的数据和控制记号的数量。

--help 打印所支持的命令行选项的说明。.

--version 显示版本号和版权。

21.2 警告选项

--warn-resources

打印（有关标准误差）以下方面的警告信息：

- 计时输入或输出操作规定之前的时间。
- 在处理器输入前，重写缓冲端口的转移寄存器中的数据。

--warn-stack

打开有关可能的堆栈破损的数据。

当XC任务尝试读或写另一个任务工作空间时，XSIM打印警告。如果使用#pragma stackfunction （参见§8）或#pragma stackcalls （参见§8）规定了任务的堆栈空间，则会出现此警告。--no-warn-registers

当寄存器在写之前读时，不会发出此警告。

21.3 追踪选项

--trace
-t 打开所有展开图的指令追踪（参见图42）。

--trace-to *file*
打开所有开展图的指令追踪。追踪为*file*的输入。

--disable-rom-tracing
关闭从ROM执行的所有指令的追踪。

--enable-fnop-tracing
打开FNOP指令的追踪。

图42： 追踪XS1处理器的输出

展开图	核心状态											地址		指令		Mem	周期
从XN得到的名称	I0	I1	I2	S0 S1 （T0 ） .. S0 S1 （T*n* ）			.	M	S	K	N	PC	（ sym + 偏移 ）：	名称	操作数	地址	@val
	- D	* P	- d	- a A i I p m s w	- b i e	*n* 状态对		- m	- s	- k	- n				val *rn* （val） res[id]	L[adr] S[adr]	

I0 ：-	无调试中断	S1 ：-	中断并失活事件
I0 ：D	导致调试中断的指令	S1 ：b	中断并激活事件
I1 : *	预期指令	S1 : **i**	激活中断且事件失活
I1 : P	暂停指令	S1 : e	中断失活且事件激活
I2 : –	未处于调试模式	M: –	MSYNC未设置
I2 : d	调试模式中的展开图	M: m	MSYNC设置
S0 : –	未使用的核心	S: –	SSYNC未设置
S0 : a	核心激活	S: s	SSYNC设置
S0 : A	核心激活（正在追踪的指令属于此核心）	K: –	INK未设置
S0 : **i**	使用ININT比特设置激活核心	K: k	INK设置
S0 : **I**	使用ININT比特设置激活核心（属于此核心）	N: –	INENB未设置
S0 : p	因为指令提取暂停的核心	N: n	INENB设置
S0 : m	使用MSYNC比特设置暂停的核心	**r*n*** （val ）	寄存器*n*的数值
S0 : s	使用SSYN比特设置暂停的核心	res[id]	资源识别符
S0 : w	使用WAITING比特设置暂停的核心	L/S[adr]	从某地址加载/贮存到某地址

XMOS

--vcd-tracing *args*

激活信号追踪。追踪数据的输出为标准VCD文件格式。

如果*args*中含有任何空格，则必须附在标记中。其格式为：

global-options*opt* *h*-tile name *h*trace-options*i* *i*

全局选项为：

-pads 打开垫追踪。

-o *file* 将输出放入*file*中。

追踪选项是与XN核心减速名称有关的展开图特有的，例如展开图[0]。

追踪选项为：

-ports 打开端口追踪。

-ports-detailed

打开更详细的端口追踪。

-cycles 打开时钟循环追踪。

-clock-blocks 打开时钟模块追踪。

-cores 打开逻辑核心追踪。

-instructions 打开指令追踪。

从不同的节点、展开图或逻辑核心输出追踪到不同的文件时，可以给此选项规定多个时间。

例如，以下命令配置模拟器以追踪展开图[0]的端口到文件追踪.vcd。

```
▶ xsim a.xe --vcd-tracing "-o trace.vcd -start-disabled -tile tile[0]
  -ports"
```

VCD 插入的追踪可以使用_traceStart（）和_traceStop（）系统调用激活和失活。-start-disabled 参数从开始失活 vcd 追踪，使用户只能激活/失活需要追踪的代码部分。例如：

```
#include <xs1.h>
#include <syscall.h>

port p1 = XS1_PORT_1A;

int main() {
    p1 <: 1;
    p1 <: 0;

    _traceStart();
```

```
   p1 <: 1;
   p1 <: 0;
   _traceStop();

   p1 <: 1;
   p1 <: 0;

   return 0;
}
```

21.4 回路插件选项

XMOS Loopback plugin 配置将要连接到一起的任何两个端口。插入的参数格式为：

-pin package pin

在包数据单上使用名称规定插脚。Package 的数值必须与用于汇编程序的 XN 文件的包节点的 ID 属性匹配（参见§46.3）。

-port name n offset

规定与提名端口对应的 n 插脚。

Name 的数值必须与用于汇编程序的 XN 文件的端口节点的 Name 属性匹配（参见§42.4.2）。

设置偏移为非零数值以规定可用插脚的子集。

-port tile p n offset

规定连接到展开图上端口 p 的 n 个插脚。

展开图的数值必须与用于汇编程序的 XN 文件的展开图节点的参考属性匹配（参见§42.4.1）。

P 可以为<xs1.h>定义的任何端口识别符。设置 offset 为非零数值，规定可用插脚的子集。

plugin 选项成对规定，连接的两端各一个。例如，以下命令配置模拟器，以使插脚连接到展开图[0]的端口 XS1_PORT_1A、到程序中端口 UART_TX 定义的插脚。

```
▶ xsim uart.xe --plugin LoopbackPort.dll '-port tile[0] XS1_PORT_1A 1 0 -port UART_TX
  1 0'
```

21.5 xSCOPE选项

--xscope args
激活xSCOPE.文件格式。
如果args中含有任何空格，则必须附在引证中。必须选择下面2个选项之一：
-offline <filename>
在xSCOPE处于离线模式下运行，并将xSCOPE输出放入给定文件中。
-realtime <URL：port>
在xSCOPE出于实时模式下运行，以给定URL端口发送xSCOPE输出：port.
以下参数可选：
-limit <num records>
限制xSCOPE输出记录到给定的数量。

例如，在xSCOPE激活离线模式时，以下将运行xSIM：
· xsim app.xe --xscope "-offline xscope.xmt"
例如，在xSIM激活实时模式时，以下将运行xSIM：
· xsim app.xe --xscope "-realtime localhost：12345"

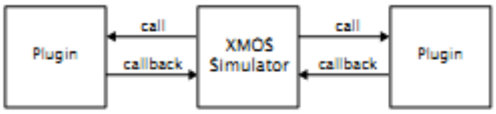
XMOS

22 XSIM测试台和插件界面

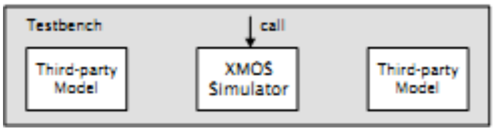
本章内容：
· 运行插件
· 插件通知
· 执行测试台
· 插件应用程序界面
· 测试台应用程序界面

XMOS仿真器提供两个接口，以便您用于在较大的系统环境中仿真您的程序。
· 插件接口使您能够连接一个或多个外部设备到仿真器上，并使用仿真器给所有外部设备计时，如图43所示。
· 测试台接口允许您连接一个或多个仿真器和外部设备，所有均在测试台的控制下计时，如图44所示。

图43：插件接口



插件；调用，；回调；XMOS仿真器；调用，回调；插件
图44：测试台接口



测试台，第三方模型；调用；XMOS仿真器；第三方模型

22.1 运行插件

插件接口使您能够写与XMOS仿真器连接的外部设备的型号。下述插件实例调回仿真器中的两个接口类型。

```
#include "xsiplugin.h"
#include <assert.h>

XsiCallbacks \*g_xsi;

XsiStatus plugin_create(void \**instance, XsiCallbacks \*xsi,
                        const char \*arguments) {
  assert(CHECK_INTERFACE_VERSION(xsi));
  g_xsi = xsi;
  return XSI_STATUS_OK;
}

XsiStatus plugin_clock(void \*instance) {
  unsigned value   = 0;
  XsiStatus status = g_xsi->sample_pin("0", "X1D52", &value);
  \*instance = (void \*)0;
  if (status == XSI_STATUS_OK)
    status = g_xsi->drive_pin("0", "X1D53", value);
  return status;
}
```

仿真器在启动时调用plugin_create。调用CHECK_INTERFACE_VERSION以检查仿真器所支持的接口版本是否与用于编译插件的接口库版本兼容。参数xsi为用于实施调回仿真器的函数指针的结构。它被分配到全局变量g_xsi中，以便以后使用。

处理器时钟每过一秒，仿真器调用一次plugin_clock。每次调用时，插件采集 [package 0, pin X1D52]的数值，并将之送到[package 0, pin X1D53]，有效的循环回两个接口类型中。包名称来自XN文件，以在仿真器上运行程序，而接口类型名称来自包数据单。

使用工具分布实例循环回插件。可以在src/plugins/ExamplePlugin/中找到。使用此插件的xc程序实例可以在examples/ExamplePlugin测试中找到。

22.2 插件通知

此通知机制允许您直接通过系统调集，在插件与xCORE窗口并联中运行的代码之间通信。下面显示了一个实例：

```
#include <syscall.h>

#define TYPE_DISPLAY 0

int main() {
    _plugins(TYPE_DISPLAY, 42, 99);
    return 0;
}
```

```
#include <stdio.h>

#define TYPE_DISPLAY 0

XsiStatus plugin_notify(void \*instance,
  int type, unsigned arg1, unsigned arg2) {

  if (type == TYPE_DISPLAY)
    printf("Args: %\%d%, %\%d%, %\%d%\n", arg1, arg2);

  return XSI_STATUS_OK;
}
```

在上述实例中，在xCORE仿真器上运行的程序调用带有用户定义通知类型和两个参数的函数 _plugins。此调用导致仿真器调用有这些参数的plugin_notify。在这种情况下，此通知函

XMOS

数仅输出参数到控制台。

22.3 执行测试台

测试台接口允许您在较大的系统环境内将XMOS仿真器实例化。此测试台负责给整个系统计时。下面的实例测试台将XMOS系统实例化并给之计时，直到完成。

```
#include "xsidevice.h"
#include <assert.h>

void *xsim = 0;

int main(int argc, char **argv) {

  XsiStatus status = xsi_create(&xsim, "test.xe");
  assert(status == XSI_STATUS_OK);

  while (status != XSI_STATUS_DONE) {
    status = xsi_clock(xsim);
    assert(status == XSI_STATUS_OK
       || status == XSI_STATUS_DONE);
  }

  status = xsi_terminate(xsim);
  assert(status == XSI_STATUS_OK);
  return 0;
}
```

测试台调用xsi_create为给定二进制定义的平台创建一个XMOS仿真器实例。在当型循环的一次迭代中，测试台调用xsi_clock，后者使仿真器通过时钟的一次滴答，执行一个或多个逻辑核心。

使用工具分布实例测试台。可以在工具安装目录src/testbenches/ExampleTestbench/中找到。examples/ExampleTestbenchTest给出了在该设备上运行的xc 程序的实例。

在您的主PC上编译插件需要使用C89 编译器。在Windows系统上，需要链接的库是lib/xsidevice.dll，而在Mac和Linux系统上，需要链接的库是lib/libxsidevice.so。

22.4 插件应用程序界面

所有插件均应包含标题文件xsiplugin.h。除非另有说明，函数成功运行时会返回XSI_STATUS_OK。

您应提供实施乙烯函数以创建您的插件，给之计时，并终止。

XsiStatus plugin_create（void **instance,
XsiCallbacks *xsi,
const char *arguments）

启动时，仿真器调用此函数。如果需要多个插件，则实例扩作为实现多实例化的手段（例如通过定义为含有插件的每个实例的状态的全局结构阵列）。Xsi含有回调函数指针的结构，而插件可将之用于与仿真器连接。含有指令行参数串的参数传递到插件。出现错误时，您可以返回XSI_STATUS_INVALID_ARGS。

XsiStatus plugin_clock（void *instance）

默认方式为，每次增加处理器时钟时，仿真器调用此函数。可以调用函数set_mhz改变时钟频率。

XsiStatus plugin_notify（void *instance,
int type,
unsigned arg1,
unsigned arg2）
当程序执行类型OS_PLUGINS的系统调用时，仿真器调用此函数。这允许目标代码与插件沟通。
XsiStatus plugin_terminate（void *instance）
在终止前，仿真器立即调用此函数。
被传递到plugin_create的结构提供的回调函数被用于配置仿真器。

22.4.1 与仿真器连接

在调用plugin_create时，仿真器提供回调函数结构（类型XsiCallbacks的参数），以用于与仿真器连接。
提供以下回调函数以配置仿真器。
XsiStatus CHECK_INTERFACE_VERSION（XsiCallbacks *）
CHECK_INTERFACE_VERSION检查在仿真器上运行的接口库的版本是否与用于编译插件的接口版本兼容。
XsiStatus （*set_mhz）（double mhz）
set_mhz设置插件计时速率，单位为MHz。仿真器有一个机制确保所有处理器和外部插件均被依照相对最快组件（及另一个）来说合适的速率计时（超时和平均）。
XsiStatus （*reset）（XsiResetType type）
reset执行设备重新设置操作。唯一支持的数值类型为XSI_RESET_HARD，它在目标平台执行所有设备的硬性重置。此影响与驱动包RESET接口类型相同。

XsiStatus （*save_state）（const char *file）
save_state将仿真器状态写入文件中。如果无法打开文件以写入，则仿真器返回XSI_STATUS_INVALID_FILE。
XsiStatus （*restore_state）（const char *file）
restore_state读仿真器状态文件，并继续从此点执行。如果文件无法打开，则仿真器返回XSI_STATUS_INVALID_FILE。
在以下函数中：

- 参数包的数值必须与用于编译运行程序的XN文件夹中的包节点的ID属性匹配。
- 接口类型的数值必须与相应的包数据单中规定的I/O接口类型的名称匹配。出现错误时，仿真器返回XSI_STATUS_INVALID_PACKAGE。
- The value of 参数窗口并联的数值必须与用于编译运行程序的XN文件夹中的窗口并联引用节点的引用属性匹配。出现错误时，仿真器返回XSI_STATUS_INVALID_TILE。
- If parameters for both如果提供了用于窗口并联和端口的参数，窗口并联可能无效，这时事件端口必须与用于编译运行程序的XN文件夹中的端口节点的名称属性匹配。否则，端口必须为格式为XS1_PORT_xy的有效端口识别符。如果规定的端口不正确，则仿真器返回XSI_STATUS_INVALID_PORT。

XMOS

· 如果用于贮存采样数值的参数无效，则仿真器返回XSI_STATUS_NULL_ARG。下面的回调函数被用于与端口连接。

XsiStatus （*sample_pin）（const char *package，const char *pin，unsigned *var）

sample_pin读规定的接口型的数值，并将之分配给一个变量：如果之前插件在驱动该接口类型，则停止驱动。

XsiStatus （*sample_port_pins）（const char *tile,
const char *port,
XsiPWord mask,
XsiPWord *var）

sample_port_pins读零个或更多个、使用掩码规定的接口类型的数值，并将之分配给一个变量：如果掩码中比特n为1，则采集连接到端口比特n的接口类型，且如果插件之前驱动接口类型，则它停止驱动；否则，该接口类型不执行行动。

XsiStatus （*is_pin_driving）（const char *package, const char *pin, int *var）

如果xCORE处理器现在正在驱动接口类型，则is_pin_driving分配给变量的数值为1，否则为0。

XsiStatus （*is_port_pins_driving）（const char *tile,
const char *port,
XsiPWord *var）

如果xCORE处理器正在驱动连接到规定端口的比特n的接口类型，则is_port_pins_driving分配给一个变量一个数值，使该数值比特n的数值为1；如果未驱动该接口类型，则比特数值为0。

XsiStatus （*drive_pin）（const char *package, const char *pin, unsigned value）

drive_pin将一个数值写入规定的接口类型。此数值留在该接口类型，直到后续调用采样或驱动该接口类型的。

XsiStatus （*drive_port_pins）（const char *tile,
const char *port,
XsiPWord mask,
XsiPWord value）

drive_port_pins写入掩码规定的零个或多个接口类型：如果掩码的比特n为1，则数值的比特n写连接到规定端口的比特n的接口类型，否则该接口类型不执行任何行动。写入每个接口类型的数值留在该接口类型，直到之后调用采样或驱动该接口类型的函数。

以下回调函数被用于与内存和除错寄存器连接。

XsiStatus （*read_mem）（const char *tile,
XsiWord32 address,
unsigned num_bytes,
unsigned char *data）

read_mem从xCORE窗口并联内存（从规定地址开始）读大量字节，并分配给它们一个变量。数据指明的数据大小必须至少与num_bytes一样大。如果内存地址无效，则仿真器返回

XMOS

XSI_STATUS_MEMORY_ERROR。

XsiStatus （*write_mem）（const char *tile,
XsiWord32 address,
unsigned num_bytes,
const unsigned char *data）
write_mem read_mem从规定的地址开始写大量字节到xCORE窗口并联内存中。数据指明的数据大小必须至少与num_bytes一样大。如果内存地址无效，则仿真器返回XSI_STATUS_MEMORY_ERROR。

XsiStatus （*read_pswitch_reg）（const char *tile,
unsigned reg_num,
unsigned *var）
read_pswitch_reg读规定的处理器开关寄存器，并将之写入一个变量中。如果寄存器编号地址无效，则仿真器返回XSI_STATUS_PSWITCH_ERROR。
XsiStatus （*write_pswitch_reg）（const char *tile,
unsigned reg_num,
unsigned value）

write_pswitch_reg写一个数值到规定的处理器开关寄存器中。如果寄存器编号地址无效，则仿真器返回XSI_STATUS_PSWITCH_ERROR。

22.5 测试台应用程序界面
测试台应包括标题文件xsidevice.h。除非另有说明，成功时函数返回XSI_STATUS_OK。
以下函数用于创建、配置、计时和终止XMOS仿真器的一个实例。
XsiStatus xsi_create（void **instance, const char *arguments）
xsi_create创建一个配备有规定的指令行参数的仿真器实例。
错误返回代码为XSI_STATUS_INVALID_ARGS。
XsiStatus xsi_clock（void *instance）
xsi_clock使系统中最快的处理器前进一个时钟循环，且可能使一个或更多个较慢的处理器也前进一个时钟循环。仿真器有一个确保所有处理器均依照相对来说正确的速率计时（超时和平均）的机制。
XSI_STATUS_OK的返回数值表明时钟操作完成，且XSI_STATUS_DONE的数值表明整个仿真已经完成。
XsiStatus xsi_reset（void *instance, XsiResetType 类型）
xsi_re设置执行设备重置操作。当前唯一支持的数值类型为XSI_RESET_HARD，在目标平台对所有设备进行硬性重置。此影响与驱动包重置接口类型相同。
XsiStatus xsi_save_state（const char *file）
xsi_save_state将仿真器状态写入文件中。如果写入的文件无法打开，则函数返回
XSI_STATUS_INVALID_FILE。

XMOS

XsiStatus restore_state（const char *file）
xsi_restore_state读仿真器状态文件并继续从此点执行。如果文件无法打开，则函数返回XSI_STATUS_INVALID_FILE。
XsiStatus xsi_terminate（void *instance）
xsi_terminate终止规定的仿真实例。

22.5.1 与仿真器的连接
以下函数可用于与仿真中的设备连接。在以下函数中：
· 如果仿真器实例无效，则函数返回XSI_STATUS_INVALID_INSTANCE。
· 参数包的数值必须与用于编译运行程序的XN文件中的包节点的ID属性匹配。
· 接口类型的数值必须与规定在相应的包数据单上的I/O接口的名称匹配。出现错误时，函数返回XSI_STATUS_INVALID_PACKAGE。
· The value of parameter参数窗口并联的数值必须与XN用于编译运行程序的XN文件中窗口并联引用节点的引用属性匹配。出现错误时，函数返回XSI_STATUS_INVALID_TILE。
· 如果提供窗口并联和端口的参数，则窗口并联可能无效，在这种情况下端口必须与用于编译运行程序的XN文件中的端口节点的名称属性匹配。否则，端口必须为有效的端口识别符，格式为XS1_PORT_xy。如果端口规定正确，则函数返回XSI_STATUS_INVALID_PORT。
· 如果用于贮存采样数值的参数无效，则函数返回XSI_STATUS_NULL_ARG。以下函数被用于与端口连接。

XsiStatus *xsi_sample_pin（void *instance,
const char *package,
const char *pin,
unsigned *var）
xsi_sample_pin读规定接口类型的数值，并将之写入变量中。如果之前测试台驱动接口类型，则它停止驱动。

XsiStatus *xsi_sample_port_pins（void *instance,
const char *tile,
const char *port,
XsiPWord mask,
XsiPWord *var）
xsi_sample_port_pins读掩码规定的零个或更多个接口类型的数值，并分配给一个变量：如果掩码的比特n为1，则连接到该端口的比特n的接口类型被采样，且如果测试台之前驱动接口类型，则它停止驱动；否则不在该接口类型上执行行动。

XsiStatus xsi_is_pin_driving（void *instance,
const char *package, const char *pin,
int *var）
如果xCORE窗口并联现在正在驱动接口类型，则xsi_is_pin_driving分配给变量的数值为1，

否则为0。

XsiStatus *xsi_is_port_pins_driving（void *instance,
const char *tile,
const char *port,
XsiPWord mask,
XsiPWord *var）

xsi_is_port_pins_driving分配给变量一个数值，从而使数值的比特n为1，而xCORE窗口并联现在驱动连接到规定端口的比特n的接口类型；比特值0强调该接口类型现在正在被驱动。

XsiStatus *xsi_drive_pin（void *instance,
const char *package,
const char *pin,
unsigned value）

xsi_drive_pin写将一个数值写入规定的接口类型。此数值保留在该接口类型上，直到后续调用采样或驱动该接口类型的函数。

XsiStatus xsi_drive_port_pins（void *instance,
const char *tile,
const char *port,
XsiPWord mask,
XsiPWord value）

xsi_drive_port_pins写入掩码规定的零个或多个接口类型：如果该掩码的比特n为1，则该数值的比特n写入与规定端口连接的比特n上，否则插脚不执行行动。写入每个接口类型的数值留在该接口类型，直到后续调用采样或驱动该插脚的函数。如果出现错误，返回代码为XSI_STATUS_INVALID_PIN和XSI_INVALID_PORT。

以下函数被用于与内存和除错寄存器连接。

XsiStatus xsi_read_mem（void *instance,
const char *tile,
XsiWord32 address,
unsigned num_bytes,
unsigned char *data）

xsi_read_mem从xCORE窗口并联内存（从规定地址开始）读大量字节，并分配给它们一个变量。数据指明的数据大小必须至少与num_bytes一样大。如果内存地址无效，则仿真器返回XSI_STATUS_MEMORY_ERROR。

XsiStatus xsi_write_mem（void *instance,
char *tile,
XsiWord32 address,
unsigned num_bytes,
const unsigned char *data）

xsi_write_mem从规定的地址开始写大量字节到xCORE窗口并联内存中。数据指明的数据大小必须至少与num_bytes一样大。如果内存地址无效，则仿真器返回XSI_STATUS_MEMORY_ERROR。

XsiStatus xsi_read_pswitch_reg（void *instance,
const char *tile,
unsigned reg_num,
unsigned *var）

xsi_read_pswitch_reg读规定的处理器开关寄存器，并将之写入变量中。如果寄存器编号地址无效，则函数返回XSI_STATUS_PSWITCH_ERROR。

XsiStatus xsi_write_pswitch_reg（void *instance,
const char *tile,
unsigned reg_num,
unsigned value）

xsi_write_pswitch_reg写一个数值到规定的处理器开关寄存器。如果寄存器编号地址无效，则函数返回XSI_STATUS_PSWITCH_ERROR。

第 I 部分调试

内容

23 使用 xTIMEcomposer 调试程序

本章内容：

- 启动调试器
- 控制程序执行
- 检查暂停的项目
- 设定断点
- 查看反汇编代码

程序在硬件或模拟器上执行时，xCORE 调试器使你看到程序内部发生的变化，从而帮助你找到错误行为的原因。

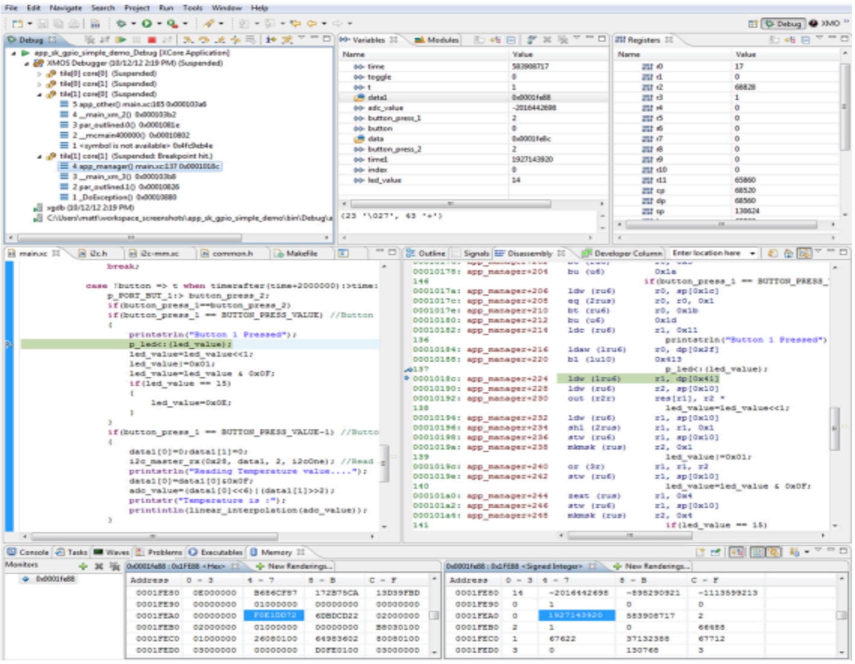


图 45.调试视图

为使程序完全可见，你必须在启用调试的前提下对其进行编译（见§9.3）。这将为你的编译器添加可执行符号，从而使调试器与源代码直接相关。注意，启用优化后再进行编译

会使调试变得更加困难。

23.1 启动调试器

在调试器控制下根据以下步骤加载程序：

1. 在“项目浏览器”中选择一个项目。

2. 点击“运行”——“调试配置”。

3. 在左侧面板中双击“XCore 应用”。xTIMEcomposer 新建新的配置并于右侧面板中显示默认设置。

4. 在“名称“文本框中输入配置名称。

5. xTIMEcomposer 试图确定可执行的目的项目。自行选择时，点击“项目”文本框右侧的“浏览”按钮，在“项目选择”对话框中选择你的项目，而后点击“搜索项目”并在“项目选择”对话框中选择可执行文件。

你必须无差错编译可执行程序，以备选择。

6. 将开发板与系统连接后，在“设备“选项面板勾选硬件选项，然后从适配器里列表中选择调试适配器。或者，勾选模拟器选项从而在模拟器上运行程序。

7. 点击“调试”以保存配置并启动调试器。如果询问你是否打开“调试“视图，则勾选“记住我的选择”并点击“是”。

xTIMEcomposer 在调试器中加载程序，并于“调试”视图中打开程序。

xTIMEcomposer 记住最近一次加载程序所用的配置。点击“调试”按钮，使用相同设置调试程序。点击“调试”按钮右边的箭头，从下拉列表中选择配置，使用不同配置。

23.2 控制程序执行

启动之后，调试器将会运行程序直到发生例外情况，或点击“暂停”按钮暂停。

点击“复位”按钮继续执行暂停的程序，或使用步骤控制，逐步进入“调试”视图中选择的核心。

进入：于“调试”视图所选择的核心中执行单线源代码。如果某一代码的下一行是函数调用，则调试器将于调用函数的第一语句处暂停，从其他代码重新开始。

 单步：在调试视图中，在所选的内核上执行单行源代码。

 单步返回：在调试视图中，在所选内核上单步进行，直到当前函数返回。如果代码的下一行是一个函数调用，则调试程序执行整个函数。所有其它内核继续执行。

 单步调试：将调试程序的语境切换到信道输出语句的对应输入内核。这对于跟踪数据路径非常有用，因为数据在内核之间流动。所有内核都不继续执行。

当调试已优化的代码时，单步运算不能保证进入源代码的下一行，因为编译器可能已经对指令重新排序，以改进性能。

23.3 检查暂停的程序

一旦程序暂停，你可以查询每一个内核的状态，并且可以检查寄存器和内存中保存的值。

检查内核的调用栈：调试视图显示了软件任务的列表，可以对每一个列表进行扩展，以显示其调用栈，如图 28 所示。

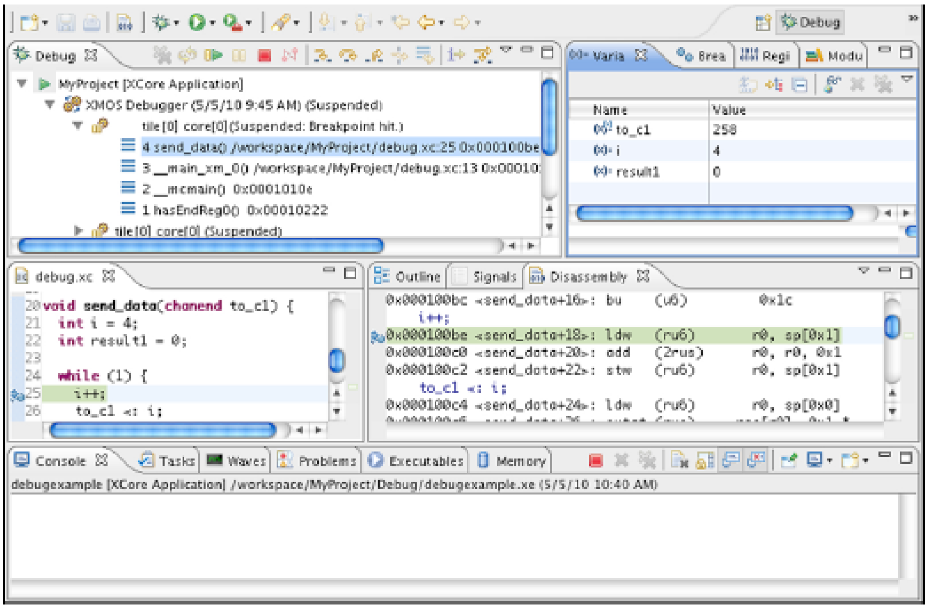


图 46：调试视图

在上例中，tile tile[0]在文件 debug.xc 的 25 行中函数 send_data 的断点暂停。

检查变量：变量视图显示了变量和它们的值。在调试视图中，点击内核调用栈中的任何函数，以查看其变量，如图 20.3 所示。

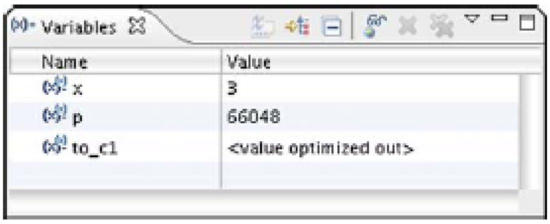


图 29：

变量视图

为了查看全局变量，右键单击变量视图，从弹出菜单中选择“增加全局变量”打开一个对话框，然后选择增加到视图中的全局变量。

不经优化编译程序保证每一个变量在其作用域的时间段内保存在内存中，从而总是能够显示它的值。如果启用了优化，则可能无法对变量进行检查，从而产生了消息<优化后的值输出>。

你可以对变量进行以下操作：

以十六进制格式显示变量的值：右键单击一个变量，以显示一个菜单，然后选择 格式▶十六进制。你也可以选择二进制，十进制或普通格式。普通格式由变量的类型确定。

改变变量的值：点击值使其高亮显示，输入一个新的值，然后按下回车。表项以黄色高亮显示，从而表明其值已经改变。这允许你按照“what-if”脚本测试发生了什么。

防止调试程序读出变量：右键单击变量，并从语境菜单中选择禁用。这在变量类型具有“易失性”时很有用。为了一次将设置应用到多个变量，在按下 Ctrl（Windows，Linux）或⌘（Mac）的同时点击多个变量，然后右键单击和从语境菜单中选择一个选项。

检查内存：内存视图提供了内存监视器列表，每一个类表代表一段内存。为了打开内存视图，选择 窗口▶显示视图▶内存。在调试视图中，点击任意内核查看内存的内容，如图 20.3 中所示。

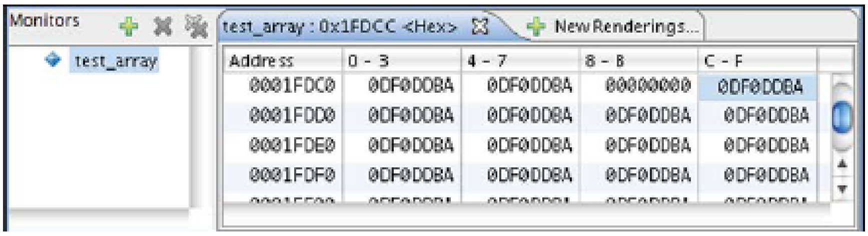


图 48：内存视图

为了指定待查看的内存位置，点击增加按钮打开内存监视器对话框，输入一个内存位置，然后点击确定。你可以输入一个绝对值或者一个 C/XC 表达式。如果要查看数组内容，只需要输入数组名称。

为了以不同格式显示内存内容（例如 Hex 或 ASCII），点击“新的释义”标签，选择一种格式，然后点击“增加释义”。xTIMEcomposer 在内存视图右侧的面板中增加了新标签，每一个标签显示了内存中值的不同解释。

23.4 设置断点

XMOS

断点时程序中的一个标记，它命令调试程序中断执行，从而使你能够调查程序的状态。你可以对任何可执行的代码行增加一个断点，从而使执行在该代码行执行之前暂停。

为了增加一个断点，在你希望暂停执行的代码行旁边的代码编辑器左边中的标记栏双击，以暂停执行。一个蓝点将会出现，代表存在断点。请注意，断点应用到执行函数的每一个内核。

断点也显示在断点视图中。为了打开断点视图，选择 窗口▶显示▶视图▶断点。在一个断点上双击确定对应代码行在源代码编辑器中的位置。

以下是你可以对断点进行的其它操作：

设置一个条件断点：右键单击断点标记，以显示一个语境菜单，然后选择“断点性质”显示性质对话框。在左侧面板中双击“通用”选项，并在右侧面板的“条件”文本框中输入 C/XC 条件表达式。表达式可以含有断点作用域内的任何变量。

设置一个条件断点：右键单击断点标记，以显示一个语境菜单，然后选择“断点性质”显示性质对话框。在左侧面板中双击“通用”选项，并在右侧面板的“条件”文本框中输入 C/XC 条件表达式。表达式可以含有断点作用域内的任何变量。

对全局变量设置一个监视点：监视点是一个特殊的断点，它在表达式的值改变时暂停执行（没有指定何时发生）。在断点视图的任意位置右键单击，然后从语境菜单中选择“增加监视点 C/XC”。在对话框中输入一个 C/XC 表达式，例如，[MAX]。选择“写入”以在表达式被写入时中断，选择“读出”以在表达式被读出时中断。

显示一个断点：在断点视图中，取消断点旁边的选择框。勾选该选择框，以重新启用断点。

移除一个断点：在代码编辑器中双击一个断点标记将其移除。或者，在断点视图中右键单击一个断点，然后从语境菜单中选择移除；为了移除所有断点，选择全部移除。

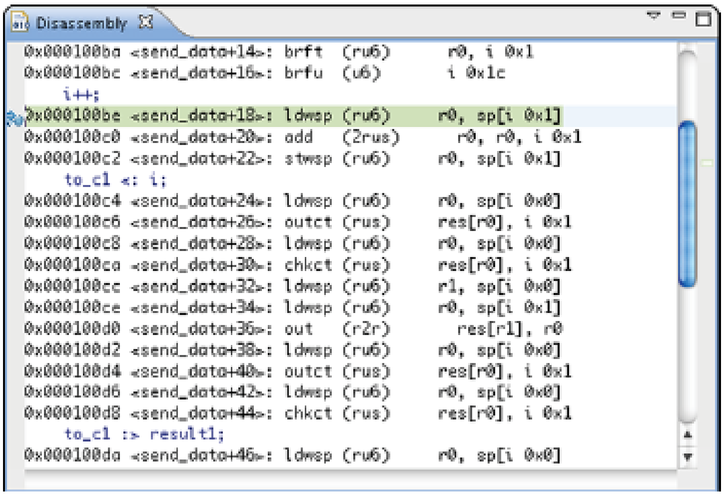
23.5 查看反汇编代码

反汇编视图显示了在目的平台上执行的汇编指令。为了打开反汇编视图，选择 窗口▶显示视图▶反汇编。

XMOS

图 49：

反汇编视图



当焦点位于反汇编视图时，xTIMEcomposer 自动启用指令步进模式。或者点击“指令步进模式”按钮启用。一旦启用，点击“步进”按钮按每一条汇编指令运行程序。

24 通过 printf 实时调试

本章内容

将 stdout 和 stderr 重定向到 XTAG-2

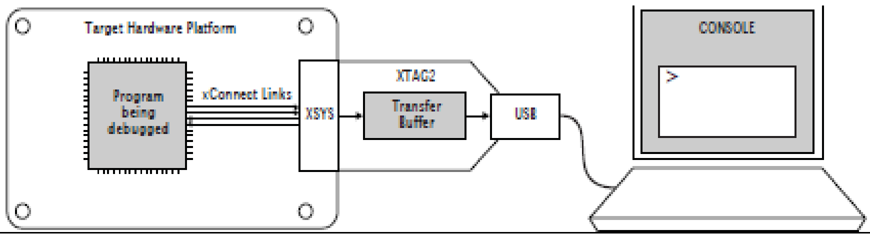
在启用 XTAG-2 输出的情况下运行程序

使用 UART 接口输出

xCORE 调试程序允许你暂停程序的执行，以便分析其内部状态。但是，如果你的程序含有时序-关键行为，例如由于其实现一个实时通信协议，暂停程序的行为可能引起其它系统组件出现故障，从而妨碍了进一步调试。

调试的一个备选方法是对你的程序增加追踪语句，以便在实时环境下观察其内部行为（有时候被称为 printf 调试）。通过打印中间计算的结果，你可以快速的隔离你的程序中出错的地方。

在传统的调试环境中，使用诸如 JTAC 标准输出数据造成妨碍内核执行的中断，从而使你的程序运行速度显著降低。xTIMEcomposer 允许你将标准流 stdout 和 stderr 重定向到 XTAG-2 调试适配器，数据缓存在调试适配器中，直到可以将其输出到主机为止。

图 50： XTAG-2 具有输入/输出重定向的调试配置	
	在此配置中，对输出例行程序的调用（例如 printf）在数据被输出到 xConnect 链接时完成，从而将对程序时序特性的影响降到最低。这允许将调试语句增加到多个时序-关键代码段，并在执行期间在控制台查看。在程序崩溃的情况下，XTAG-2 缓冲区中所有剩下的内容被发送到 PC，从而确保重要信息不会丢失。 如果你使用了 legacy FTDI 或 XTAG-1 调试适配器，或者如果你的目的硬件上的 XSYS 连接器没有提供 xConnect 链接，则你可以通过你的适配器的 UART 接口（请见 21.3）输出数据。请注意，UART 接口严重影响性能。

24.1 将 stdout 和 stderr 重定向到 XTAG-2

以下程序将标准输出重定向到 XTAG-2。

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

port receive;
port transmit;

int process(int);

void xscope_user_init(void) {
  xscope_register(0);
  xscope_config_io(XSCOPE_IO_BASIC);
}

int main() {

  while (1) {
    int dataIn, dataOut;

    receive :> dataIn;
    dataOut = process(dataIn);

    /* Debug Information */
    if (dataOut < 0)
      printf("%d %d", dataIn, dataOut);

    transmit <: dataOut;
  }
}
```

在构造符 xscope_user_init 中，对 xscope_register 的调用初始化了 XTAG-2 接口，然后对 xscope_config_io 的调用将流 stdout 和 stderr 重定向到此接口。

主程序从一个端口输出数据，在该端口执行计算，并将结果输出到另一个端口。它使用了标准输出函数 printf 记录计算结果小于零的实例。

你可以同时在任何内核上使用 C 标准输入/输出函数。这一使用引起单通道末端被分配到输出数据的每一个数据块上。

你可以通过调用 xscope_config_io（选项 XSCOPE_IO_TIMED）对输出数据加时间戳。这使得输出时间戳随数据显示在控制台中。请注意，这也减少了任何时间可以缓冲的数据量。

XMOS

24.2 在 XTAG-2 输出启用的情况下运行程序

为了将标注输出重定向到 XTAG-2 并将其显示在控制台中，你必须通过 Xscope 仪表库编译和运行你的程序。为了编译和运行你的程序，请遵守以下步骤：

1. 打开你的项目所用的 Makefile。

2. 定位你的编译配置所用的 XCC_MAP_FLAGS_config 变量，例如，XCC_MAP_FLAGS_Release。

3. 增加选项-fxscope。

4. 如果使用 XDE 开发，对你的目的设备创建一个运行配置（请见 13.1）。在 Xscope 标签中，选择“离线”模式。点击运行保存和运行此配置。

XDE 加载你的程序，同时在控制台中显示从 XTAG-2 收到的数据。

5. 如果你使用命令行工具开发，跳过选项—xscope 到 XRUN，例如：

► xrun --xscope myprog.xe

XRUN 加载你的程序，并保持到 XTAG-2 适配器的连接，同时在终端显示从其收到的数据。当程序执行对 exit 的调用时，XRUN 终止。

24.3 使用 UART 接口输出

如果你使用了 legacy FTDI 或 XTAG-1 调试适配器，或者如果你的目的硬件上的 XSYS 连接器没有提供 xConnect 链接，则你可以通过你的适配器提供的 UART 接口输出数据。

为了使用 UART 接口，你必须提供具有 1 位 UART TX 端口的 Xscope 库，通过将该端口引脚与你的调试适配器上的 UART-TX 引脚连接，对该端口初始化。下面给出了一个初始化的例子。

```
#include <platform.h>
#include <xscope.h>

port uart_tx = PORT_UART_TX;

void xscope_user_init(void) {
  xscope_register(0);
  xscope_config_uart(uart_tx);
  xscope_config_io(XSCOPE_IO_BASIC);
}
```


XMOS

为了在 XDE 中运行你的程序，对你的目的设备创建一个运行配置（请见 13.1），并选择选项“运行 UART 服务器”。

为了使用命令行工具运行你的程序，跳过选项--uart 到 XRUN，例如：

▶ xrun --uart --xscope myprog.xe

⚠ 因为 UART 接口使用了一个端口取代 xConnect 链接，你只能在单个数据块上使用 C 标准输入/输出函数。

XMOS

第 J 部分
闪存编程

25 具有闪存的设计和制造系统

本章内容

从闪存引导一个程序

生成制造所用的 flash 镜像

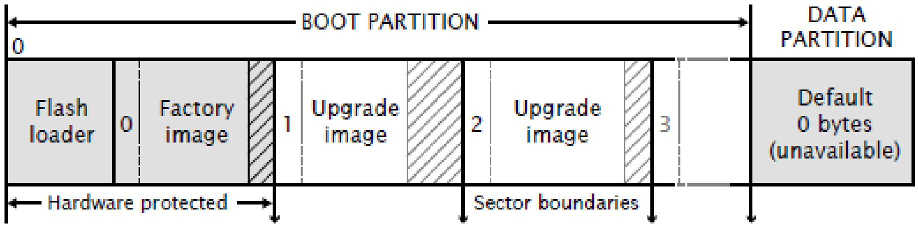
执行现场升级

配置 flash 加载器

xTIMEcomposer 可用于找到使用 SPI 闪存进行引导和长期存储的 xCORE 设备。图 33 中给出了 xCORE 闪存格式。

图 51：

闪存格式图



从逻辑上将闪存分为引导和数据分区。引导分区由闪存加载器和其后的“出厂镜像”和零个或多个可选的“升级镜像”组成。每一个镜像以一个描述符开始，描述符含有一个唯一的版本号，含有程序所用的每一个数据块的代码/数据分段列表和 CRC。在默认情况下，闪存加载器引导具有有效 CRC 的最高版本镜像。

25.1 从闪存引导程序

为了将一个程序加载到你的开发板上的 SPI 闪存装置上，启动命令行工具（请见 3.2)，并输入以下命令：

1. xflash –l

XFLASH 按以下格式打印连接到你的计算机和每一个 JTAG 链上上设备的所有 JTAG 适配器：

ID 名称 适配器 ID 设备

2. xflash --id ID program.xe

XFLASH 从你的已编译程序以 xCORE 闪存格式（含有第一个阶段加载器和由二进制和数据段构成的出厂镜像）生成镜像。然后它将此镜像写入使用

XMOS

xCORE 设备的闪存。

⚠ 用于编译你的程序的 XN 文件必须定义 SPI 闪存设备，并指定与其相连的 Xcore 设备的四个端口（请见 XM-000929-PC）。

25.2 生产一个用于制造的闪存镜像

在制造环境中，通常将相同的程序编入多个闪存设备。

为了以 Xcore 闪存格式生成镜像文件（随后可以将这种格式编程到闪存设备中），启动命令行工具（见 3.2），然后输入以下命令：

► xflash program.xe -o image-file

XFLASH 生成由第一阶段加载器和你的程序（作为出厂镜像）构成的镜像，它将该镜像写入指定的文件。

25.3 执行一个现场升级

xTIMEcomposer 和 libflash 库允许你在产品的生命周期管理多个固件升级。你可以使用 XFLASH 创建一个升级镜像（从你的程序），使用 libflash 将此镜像写入引导分区。通过使用 libflash，升级能够避免部分完整写入，例如，由于电源故障：如果引导期间升级镜像的 CRC 失败，则将加载之前的镜像。

25.3.1 写一个自我更新的程序

图 52 中的程序例子使用了 libflash 库自我更新。

对 fl_connect 的调用打开了 xCORE 和 SPI 设备之间的连接，对 fl_getPageSize 的调用确定了 SPI 设备的页面大小。所有读取和写入操作发生在页面层次。

通过调用 fl_getFactoryImage 和 getNext- BootImage 定位第一个更新页面。在定位之后，fl_startImageReplace 准备此镜像，以便由规定（最大）大小的新镜像代替。必须调用 fl_startImageReplace，直到其返回零，返回零表示准备完成。

函数 fl_writeImagePage 将数据的下一页写入 SPI 设备。对此函数的调用在数据被输出到设备之后返回，但是可能在设备已经将数据写入其闪存之前返回。这增加了处理器抓取数据下一页的时间量。函数 fl_endWriteImage 等待 SPI 设备将数据的最后一页写入其闪存。为了简化写入操作，XFLASH 对升级镜像增加了填充内容，以确保其大小是页面大小的倍数。

XMOS

图 52：

使用 libflash 自我升级的 C 程序

```
#include <platform.h>
#include <flash.h>

#define MAX_PSIZE 256

/* initializers defined in XN file
 * and available via platform.h */

fl_SPIPorts SPI = { PORT_SPI_MISO,
                    PORT_SPI_SS,
                    PORT_SPI_CLK,
                    PORT_SPI_MOSI,
                    XS1_CLKBLK_1 };

int upgrade(chanend c, int usize) {

  /* obtain an upgrade image and write
   * it to flash memory
   * error checking omitted */

  fl_BootImageInfo b;
  int page[MAX_PSIZE];
  int psize;

  fl_connect(SPI);

  psize = fl_getPageSize();
  fl_getFactoryImage(b);
  fl_getNextBootImage(b);

  while(fl_startImageReplace(b, usize))
    ;
  for (int i=0; i page[j];)
    fl_writeImagePage(page);

  fl_endWriteImage();

  fl_disconnect();

      return 0;
}

int main() {
  /* main application - calls upgrade
   * to perform an in-field upgrade */
}
```

调用 fl_disconnect 关闭了 xCORE 和 SPI 设备之间的连接。

XMOS

25.3.2 编译和部署升级器

为了编译和部署你的程序的第一个版本，启动命令行工具（请见 3.2），然后输入以下命令：

xcc file.xc -target=boardname -lflash -o first-release.xe

XCC 编译你的程序，并将其连接到 libflash。或者对你的 Makefile 增加选项-lflash。

xflash first-release.xe -o manufacture-image

XFLASH 以 Xcore 闪存格式生成一个镜像，镜像含有第一阶段加载器和你的程序的第一个版本，以作为出厂镜像。

为了编译和部署你的程序的升级版本，输入以下命令：

xcc file.xc -target=boardname -lflash -o latest-release.xe

XCC 编译你的程序，并将其连接到 libflash。

xflash --upgrade version latest-release.xe -o upgrade-image

XFLASH 生成具有指定版本号（必须大于零）的升级镜像。你的程序必须获得此镜像，以自我更新。

如果更新操作成功，则在重置设备时加载器引导更新镜像，否则其将引导出厂镜像。

25.4 优化闪存加载器

xTIMEcomposer 允许你配置选择从闪存加载哪一个镜像的机制。图 35 中的程序例子基于数据分区开始时的值确定加载哪一个镜像。

xCORE 加载器首先调用函数 init，然后反复声明引导分区中的每一个镜像。对于每一个镜像，其以镜像版本调用 checkCandidateImageVersion，然后如果函数返回值不等于零，则其 CRC 验证通过，其以镜像版本号和地址调用 recordCandidateImage。最后，加载器调用 reportSelectedImage，以获得所选镜像的地址。

为了生成一个定制加载器，你需要定义函数 init，checkCandidateImageVersion，recordCandidateImage 和 reportSelectedImage。

XMOS

加载器提供了函数 readFlashDataPage。

图 53：

定制闪存加载器的 C 函数

```
extern void *readFlashDataPage(unsigned addr);

int   dpVersion;
void *imgAdr;

int init(void) {
  void *ptr = readFlashDataPage(0);
  dpVersion = *(int *)ptr;
}

int checkCandidateImageVersion(int v) {
  return v == dpVersion;
}

void recordCandidateImage(int v, unsigned adr) {
  imgAdr = adr;
  return 1;
}

unsigned reportSelectedImage(void) {
  return imgAdr;
}
```

25.4.1 编译加载器

为了创建含有定制闪存加载器和出厂镜像的闪存镜像，启动命令行工具（请见 3.2），然后输入以下命令：

xcc -c file.xc -o loader.o

XCC 编译你用于镜像选择的函数，生成一个二进制对象。

xflash bin.xe --loader loader.o

XFLASH 将含有定制加载器和出厂镜像的闪存镜像写入规定的文件。

25.4.2 增加额外的镜像

以下命令编译了含有定制加载器、出厂镜像和两个额外镜像的闪存镜像：

► xflash factory.xe --loader loader.o --upgrade 1 usb.xe 0x20000

--upgrade 2 avb.xe

到--upgrade 的参数包括版本号、可执行文件和可选大小（单位为字节）。XFLASH 将每一个升级镜像写入下一个扇区边界。大小参数用于对镜像加入填充，从而能够在将来通过更大的镜像对其现场更新。

XMOS

26 libflash 应用程序接口

本章内容

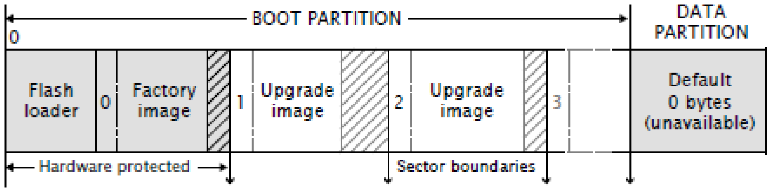
一般操作

引导分区函数

数据分区函数

Libflash 库提供了对使用下图所示的 XCORE 格式的 SPI 闪存设备读写数据的函数。

图 54：闪存格式图



所有函数的原型都在头文件<flash.h>中。除非另有规定，函数运行成功返回 0，运行失败返回 0 以外的任何数。

26.1 一般操作

在试图使用 SPI 设备之前，程序必须明确的打开到 SPI 设备的连接，并且必须在完成对设备的访问之后断开。

函数 fl_connect 和 fl_connectToDevice 需要一个类型 fl_SPIPorts 的参数，该参数定义了用于连接到该设备的四个端口和时钟块。

```
typedef struct {
  in buffered port:8 spiMISO;
  out port spiSS;
  out port spiCLK;
  out buffered port:8 spiMOSI;
  clock spiClkblk;
} fl_SPIPorts;
```

int fl_connect(fl_SPIPorts *SPI)

fl_connect 打开到指定 SPI 设备的连接。

int fl_connectToDevice(fl_SPIPorts *SPI, fl_DeviceSpec spec[], unsigned n)

fl_connectToDevice 打开到 SPI 设备的连接。它通过 n 个 SPI 设备规约的数组枚举，尝试使用每一个规约连接，直到它成功为止。

int fl_getFlashType(void)

fl_getFlashType 对闪存设备返回 enum 值。下面给出了 libflash 已知的设备枚举。

```
typedef enum {
  UNKNOWN = 0,
  ALTERA_EPCS1,
  ATMEL_AT25DF041A,
  ATMEL_AT25FS010,
  ST_M25PE10,
  ST_M25PE20,
  WINBOND_W25X40
} fl_FlashId;
```

如果函数调用 fl_connectToDevice(p, spec, n)用于连接到闪存设备，fl_getFlashType 返回参数值 spec[i].flashId，其中 i 是所连接设备的索引。

unsigned fl_getFlashSize(void)

fl_getFlashSize 返回 SPI 设备的容量（单位字节）

int fl_disconnect(void)

fl_disconnect 关闭到 SPI 设备的连接。

26.2 引导分区函数

默认情况下，引导分区的大小用于设置闪存设备的大小。通过一个枚举器接口，提供了对引导镜像的访问。

int fl_getFactoryImage(fl_BootImageInfo *bootImageInfo)

fl_getFactoryImage 提供了关于出厂引导镜像的信息。

int fl_getNextBootImage(fl_BootImageInfo *bootImageInfo)

fl_getNextBootImage 提供了关于下一个引导镜像的信息。在定位之后，就可以对镜像升级。另外还提供了读取升级镜像的函数。

unsigned fl_getImageVersion(fl_BootImageInfo *bootImageInfo)

fl_getImageVersion 返回指定镜像的版本号。

int fl_startImageReplace(fl_BootImageInfo *, unsigned maxsize)

fl_startImageReplace 准备 SPI 设备用于替换镜像。在此调用之后，无法分配旧的镜像。

试图写入数据分区或另一个升级镜像的空间是无效的。非零返回值说明准备还没有完成，并且应当再次调用此函数。此行为允许由程序屏蔽的扇区擦除的延迟。

int fl_startImageAdd(fl_BootImageInfo*, unsigned maxsize, unsigned padding)

fl_startImageAdd 准备 SPI 设备用于在指定镜像之后增加一个镜像。在之前的镜像之后，新镜像的开始至少是填充字节。

试图写入数据分区或另一个升级镜像的空间是无效的。非零返回值说明准备还没有完成，并且应当再次调用此函数。此行为允许由程序屏蔽的扇区擦除的延迟。

int fl_startImageAddAt(unsigned offset, unsigned maxsize)

fl_startImageAddAt 准备 SPI 设备用于从出厂地址之后第一个扇区的基址开始，以指定的地址偏移量增加一个镜像。

试图写入数据分区或另一个升级镜像的空间是无效的。非零返回值说明准备还没有完成，并且应当再次调用此函数。

int fl_writeImagePage(const unsigned char page[])

fl_writeImagePage 等待，直到 SPI 设备能够接受一个请求为止，然后将数据的下一页输出到设备。试图超过传递到 fl_startImageReplace, fl_startImageAdd 或 fl_startImageAddAt 的最大大小写入是无效的。

int fl_writeImageEnd(void)

fl_writeImageEnd 等待，直到 SPI 设备将数据的最后一页写入其内存。

int fl_startImageRead(fl_BootImageInfo *b)

fl_startImageRead 准备 SPI 设备用于读取指定的升级镜像的内容。

int fl_readImagePage(unsigned char page[])

fl_readImagePage 从 SPI 设备输入数据的下一页，并将其写入数组页。

int fl_deleteImage(fl_BootImageInfo* b)

fl_deleteImage 通过指定的镜像擦除升级镜像。

26.3 数据分区函数

假设所有闪存设备具有统一的页面大小，但是假设没有统一的扇区大小。读写操作发生在页面层次，并且擦除操作发生在扇区层次。这表示为了写入扇区的一部分，至少有一个扇区的缓存大小用于储存其它数据。

在以下函数中，写入数据分区和从数据分区擦除并不是故障安全的。如果操作被中断，例如由于电源故障，页面或分区中的数据是不确定的。

unsigned fl_getDataPartitionSize(void)

fl_getDataPartitionSize 返回数据分区的大小（单位为字节）

int fl_readData(unsigned offset, unsigned size, unsigned char dst[])

fl_readData 从一个偏移量将字节数读入数据分区，并将它们写入数组 dst。

unsigned fl_getWriteScratchSize(unsigned offset, unsigned size)

fl_getWriteScratchSize 返回 fl_writeData 对给定参数所需的缓存大小

int fl_writeData(unsigned offset,

unsigned size,

const unsigned char src[],

unsigned char buffer[])

fl_writeData 将数组 src 写入数据分区中的指定偏移量。它使用数组缓冲区保存必须重写的页面数据。

26.3.1 页面级别的函数

unsigned fl_getPageSize(void)

fl_getPageSize 返回页面大小（单位为字节）

unsigned fl_getNumDataPages(void)

fl_getNumDataPages 返回数据分区中页面数量

unsigned fl_writeDataPage(unsigned n, const unsigned char data[])

fl_writeDataPage 将数组数据写入数据分区中的第 n 个页面。数据数组必须至少与页面大小一样大；如果更大，则最高的元素被忽略。

unsigned fl_readDataPage(unsigned n, unsigned char data[])

fl_readDataPage 读出数据分区中的第 n 个页面，并将其写入数组数据。数据的大小必须至少与页面大小一样大。

26.3.2 扇区级别函数

unsigned fl_getNumDataSectors(void)

fl_getNumDataSectors 返回数据分区中的扇区数量。

unsigned fl_getDataSectorSize(unsigned n)

fl_getDataSectorSize 返回数据分区中第 n 个扇区的大小（单位为字节）

unsigned fl_eraseDataSector(unsigned n)

fl_eraseDataSector 擦除数据分区中的第 n 个扇区

unsigned fl_eraseAllDataSectors(void)

fl_eraseAllDataSectors 擦除数据分区中的所有扇区。

27 libflash 原生支持的设备列表

Libflash 支持市场上的大量闪存设备。使用 SPI 规约文件描述每一种闪存设备。图 37 中的表列出了通过 xTIMEcomposer 包括的 SPI 规约文件的闪存设备列表。

图 55

libflash 原生支持的闪存设备列表

制造商	部件编号	通过默认值在 libflash 中启用
Altera	EPCS1	Y
AMIC	A25L016	N
	A25L40P	N
	A25L40PT	N
	A25L40PUM	N
	A25L80P	N
Atmel	AT25DF021	N
	AT25DF041A	Y
	AT25F512	N
	AT25FS010	Y
EMT	F25L004A	N
Macronix	MX25L1005C	N
NUMONYX	M25P10	N
	M25P16	N
	M25P40	N
	M45P10E	N
SST	SST25VF010	N
	SST25VF016	N
	SST25VF040	N
ST Microelectronics	M25PE10	Y
	M25PE20	Y
Winbond	W25X10	N
	W25X20	N
	W25X40	Y

XMOS

28 增加对新闪存设备的支持

本章内容

Libflash 设备标识

页面大小和页面数

地址大小

时钟频率

读取设备标识

扇区擦除

写入启用/禁用

内存保护

编程命令

读取数据

扇区信息

状态寄存器位

增加对 xTIMEcomposer 的支持

选择闪存设备

为了支持新的闪存设备，必须写入描述设备特征的配置文件，例如页面大小，页面数量和读取、写入和擦除数据命令。这些信息可以在闪存设备的数据单中找到。市场上的许多设备都可以使用这些配置参数描述；这些还没有得到支持。

下文给出了 Numonyx M25P10-A[2]的配置文件。该设备被描述为 C 结构的初始化程序，C 结构的值在以下章节中描述。

[2] http://www.xmos.com/references/m25p10a

```
10,                        /* 1.  libflash device ID */
256,                       /* 2.  Page size */
512,                       /* 3.  Number of pages */
3,                         /* 4.  Address size */
4,                         /* 5.  Clock divider */
0x9f,                      /* 6.  RDID cmd */
0,                         /* 7.  RDID dummy bytes */
3,                         /* 8.  RDID data size in bytes */
0x202011,                  /* 9.  RDID manufacturer ID */
0xD8,                      /* 10. SE cmd */
0,                         /* 11. SE full sector erase */
0x06,                      /* 12. WREN cmd */
0x04,                      /* 13. WRDI cmd */
PROT_TYPE_SR,              /* 14. Protection type */
{{0x0c,0x0},{0,0}},        /* 15. SR protect and unprotect cmds */
0x02,                      /* 16. PP cmd */
0x0b,                      /* 17. READ cmd */
1,                         /* 18. READ dummy bytes*/
SECTOR_LAYOUT_REGULAR,     /* 19. Sector layout */
{32768,{0,{0}}},           /* 20. Sector sizes */
0x05,                      /* 21. RDSR cmd*/
0x01,                      /* 22. WRSR cmd */
0x01,                      /* 23. WIP bit mask */
```

28.1 Libflash 设备标识

```
10, /* 1. libflash device ID */
```

Libflash 在调用函数 fl_getFlashType 时需要该值，以使应用程序能够识别所连接的闪存设备。

28.2 页面大小和页面数量

```
10,                        /* 1.  libflash device ID */
256,                       /* 2.  Page size */
```

这些值规定了每一个页面的大小（单位为字节）和所有可用扇区的全部页面数量。在 M25P10-A 数据单中，可以从第 6 页的以下段落找到这些内容：

内存被组织为 4 个扇区，每一个含有 128 个页面。每一个页面的宽度为 256 字节。因此，可以将整个页面看作由 512 个页面或者 131072 个字节构成。

28.3 地址大小

```
3,                          /* 4.  Address size */
```

这些值规定了用于表示地址的字节数。图 38 复制了提供此信息的 M25P10-A 数据单的一部分。在该表中，所有指令要求地址占用三个字节。

图 56：M25P10-A 数据单 17 页上的表 4

指令	描述	单字节指令代码		地址字节	空字节	数据字节
WREN	启用写入	0000 0110	06h	0	0	0
WRDI	禁用写入	0000 0100	04h	0	0	0
RDID	读出	1001 1111	9fh	0	0	1 到 3
RDSR	读出状态寄存器	0000 0101	05h	0	0	1 到∞
WRSR	写入状态寄存器	0000 0001	01h	0	0	1
READ	读出数据字节	0000 0011	03h	3	0	1 到∞
FAST_READ	以更高的速度读出数据字节	0000 1011	08h	3	1	1 到∞
PP	页面程序	0000 0010	02h	3	0	1 到 256
SE	扇区擦除	1101 1000	D8h	3	0	0
BE	批量擦除	1100 0111	C7h	0	0	0
DP	深度低耗电	1011 1001	B9h	0	0	0
RES	从深度低耗电发布，并读出电子签名	1010 1011	abh	0	3	1 到∞
	从深度低耗电发布			0	0	0

28.4 时钟频率

```
4,                          /* 5.  Clock divider */
```

该值用于确定与 SPI 设备交互的时钟频率。对于 n 的值，所用的 SPI 时钟频率为 100/2*n MHz。libflash 支持最大 12.5MHz 的频率。

图 39 复制了提供此信息的 M25P10-A 数据单的一部分。AC 特性表给出了配置文件中所用的所有指令，正如在本文中所讨论的，其最高可以以 25MHz 运行。此频率高于 libflash 可以支持的频率，所以给出的值为 4，以产生 12.5MHz 的时钟。

总的来说，如果 SPI 设备对 libflash 所用的不同命令支持不同的时钟频率，则必须规定最低的值。

图 57：

M25P10-A 数据单 40 页上的表 18（仅有前四个表项）

符号	备选	参数	最小	类型	最大	单位
		以下指令的时钟频率：FAST_READ, PP, SE, BE,DP, RES, WREN, WRDI, RDSR, WRSR				
		READ 指令的时钟频率				
		时钟高时间				
		时钟低时间				

28.5 读取设备标识

```
0x9f,                    /* 6.  RDID cmd */
0,                       /* 7.  RDID dummy bytes */
3,                       /* 8.  RDID data size in bytes */
0x202011,                /* 9.  RDID manufacturer ID */
```

大部分的闪存设备具有硬件标识符，可用于标识该设备。当应用程序支持一个或多个闪存设备时标识符由 libflash 使用，以确定所连接的设备类型。读取设备标识的顺序通常是发出一个 RDID（读取 ID）命令，等待零个或多个空字节，然后读出一个或多个数据字节。

图 38 复制了提供此信息的 M25P10-A 数据单的一部分。指令 RDID 的列表示命令值为 0x9f，也即没有空字节，并且有一到三个空字节。如图 40 和 41 所示，读出的数据量依赖于是否只需要制造商 ID（第一字节），或者是否需要制造商 ID 和设备标识（第二字节）。为了唯一的标识该设备，需要所有三个字节，所以制造商 ID 被规定为三字节的值 0x202011。

图 58：

M25P10-A 数据单 19 页上的表 5

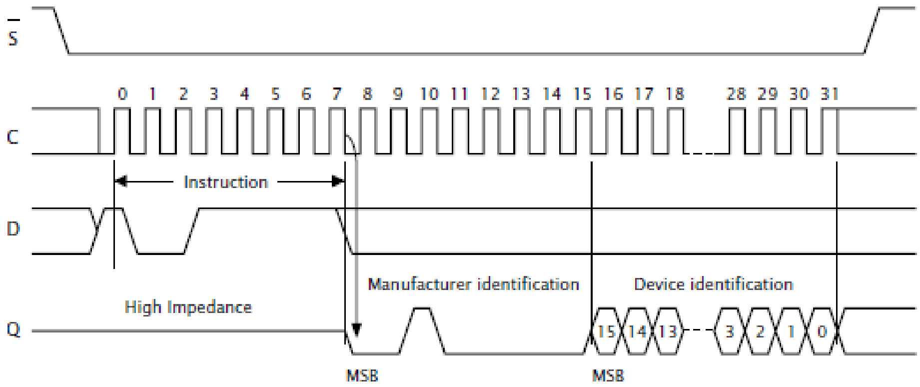
制造商标识	设备标识	
	内存类型	内存容量
20h	20h	11h

一般来说，如果有 RDID 命令的选择，则应当首选 JEDEC 兼容的命令。否则，应当使用返回最长 ID 的命令。

XMOS

图 59：

M25P10-A 数据单 19 页上的图 9



28.6 扇区擦除

```
0xD8,                   /* 10. SE cmd */
0,                      /* 11. SE full sector erase */
```

大部分闪存设备提供了擦除所有扇区或部分扇区的命令。

图 38 复制了提供了此信息的 M25P10-A 数据单的一部分。指令 SE 列显示命令值为 0xd8。在 M25P10-A 数据单上，被擦除的数据量可以从 28 页的第一段上找到：

扇区擦除（SE）指令对所选扇区内的所有位设置为 11’ (FFh)。

在此例中，SE 命令擦除了所有扇区，所以 SE 数据值被设置为 0。如果被擦除的位数小于整个扇区，则应当将该值设置为被擦除的字节数。

28.7 写入启用/禁用

```
0x06,                   /* 12. WREN cmd */
0x04,                   /* 13. WRDI cmd */
```

大部分的闪存设备提供指令启用和禁用对内存的写入。图 38 复制了提供了此信息的 M25P10-A 数据单的一部分。指令 WREN 行显示该命令值为 0x06，并且指令 WRDI 行显示该命令值为 0x04。

XMOS

28.8 内存保护

```
PROT_TYPE_SR,               /* 14. Protection type */
{{0x0c,0x0},{0,0}},         /* 15. SR protect and unprotect cmds */
```

当写入被禁用时，一些闪存设备提供了额外的扇区保护。对于支持此能力的设备，libflash 尝试保护闪存镜像，防止被应用程序意外影响。保护类型的支持值为：

PROT_TYPE_NONE

设备没有提供保护

PROT_TYPE_SR

设备通过写入状态寄存器提供保护

PROT_TYPE_SECS

设备提供命令保护单独扇区

保护细节被规定为以下命令格式的一部分：

{{a,b},{c,d}}

设备没有提供保护，所有值都应当设置为 0。如果设备提供 SR 保护，应当将 a 和 b 设置为写入 SR 的值，以启用和禁用设备保护，并将 c 和 d 设置为 0。否则，应当将 c 和 d 设置为写入到命令的值，以启用和禁用设备保护，并将 a 和 b 设置为 0。

图 42 和图 43 复制了提供了此信息的 M25P10-A 数据单的一部分。第一个表显示状态寄存器应当被设置为 1，以保护所有扇区，并且全部为 0 禁用保护。第二个表显示这些是 SR 的 2 和 3 位。

图 60

M25P10-A 数据单 13 页上的表 2

状态寄存器		内存内容	
BP1	BP0	保护区	无保护区
0	0	无	所有扇区（四个扇区：0，1，2 和 3）
0	1	前四分之一（扇区 3）	后四分之三（三个扇区：0 到 2）
1	0	前一半（两个扇区：2 和 3）	后一半（扇区 0 和 1）
1	1	所有扇区（四个扇区：0，1，2 和 3）	无

图 61

M25P10-A 数据单 20 页上的表 6

b7							b0
SRWD	0	0	0	BP1	BP0	WEL	WIP

状态寄存器写保护

块保护位

启用写入开关位

写入进度位

28.9 编程命令

图 62 : F25L004A 数据单第 12 页的表 7

```
0x02,                    /* 16. PP cmd */
```

每次编程时，将设备编程为页面或数个子节。如果有页面编程则应使用，因为它总结了通过 SPI 接口传输的数据的数量。

图 38 为提供此信息的 M25P10-A 数据单的一部分。表中提供了一个页面程序命令，且其数值为 0x02。

如果不支持页面编程，则此数值为三个单独的数值的级联。比特 0..7 必须设置为 0。比特 8..15 应含有程序命令。比特 16..23 应含有每个命令的子节数。Libflash 库要求第一个程序命令接受三字节地址，但是后续的程序命令使用自动地址增量（AAI）。

无 PP 命令的设备实例为 ESMT F25L004A[3]。图 44 为提供此信息的数据单 F25L004A 的一部分。在计时图中，该 AAI 命令的数值为 0xad，后面为三字节地址和两子节数据。

符号	参数	最小值	单位
TPU-READ	VDD Min to Read Operation	10	µs
TPU-WRITE	VDD Min to Write Operation	10	µs

图 44 F25L004A 数据表第 12 页表 7

规范文件中的相应条目为：

```
0x00|(0xad<<8)|(2<<16), /* No PP, have AAI for 2 bytes */
```

[3] http://www.xmos.com/references/f25l004

28.10 读数据

```
0x0b,                   /* 17. READ cmd */
1,                      /* 18. READ dummy bytes*/
```

从设备读数据的次序一般为发布 READ 命令、等待清零或更多虚拟子节，然后读一个或更多个数据子节。

图 38 为提供此信息的数据单 M25P10-A 的一部分。有两个可用于读数据的命令：READ 和 FAST_READ。指令 FAST_READ 的行显示命令数值为 0x0b，后面为一个虚拟子节。

28.11 矢量信息

```
SECTOR_LAYOUT_REGULAR,  /* 19. Sector layout */
{32768,{0,{0}}},        /* 20. Sector sizes */
```

第一个数值规定了所有矢量的大小是否相同。所支持的数值为：

SECTOR_LAYOUT_REGULAR

矢量的大小相同

SECTOR_LAYOUT_IRREGULAR

矢量的大小不同

在 M25P10-A 数据单中可以从第 15 页的以下段落中找到这些：

内存组织为：

· 131,072 子节（每个 8 比特）

· 4 个矢量（256 千比特，每个 32768 子节）

· 512 页（每个 256 子节）

矢量大小规定为结构的一部分：{a, {b, {c}}}。如果是常规矢量大小，则大小规定在 a 中。b 和 c 的数值应为 0。

如果是非常规的矢量大小，则矢量的大小规定在 b 中。每个矢量的页面编号的对数（底数为 2）规定在 c 中。数值 a 应为 0。具有非常规矢量的设备的实例为 AMIC A25L80P[4]。图 45 为提供矢量信息的数据单的一部分。

[5] http://www.xmos.com/references/a25l80p

XMOS

图 63：A25L80P 数据单第 7 页的表 2

扇区	扇区大小 （Kb）	地址域	
15	64	F0000h	FFFFFh
14	64	E0000h	EFFFFh
13	64	D0000h	DFFFFh
12	64	C0000h	CFFFFh
11	64	B0000h	BFFFFh
10	64	A0000h	AFFFFh
9	64	90000h	9FFFFh
8	64	80000h	8FFFFh
7	64	70000h	7FFFFh
6	64	60000h	6FFFFh
5	64	50000h	5FFFFh
4	64	40000h	4FFFFh
3	64	30000h	3FFFFh
2	64	20000h	2FFFFh
1	64	10000h	1FFFFh
0-4	32	08000h	0FFFFh
0-3	16	04000h	07FFFh
0-2	8	02000h	03FFFh
0-1	4	01000h	01FFFh
0-0	4	00000h	00FFFh

规范文件中相应的条目为：

```
SECTOR_LAYOUT_IRREGULAR,
    {0,{20,{4,4,5,6,7,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8}}},
```

28.12 状态寄存器比特

```
0x05,                    /* 21. RDSR cmd*/
0x01,                    /* 22. WRSR cmd */
0x01,                    /* 23. WIP bit mask */
```

大部分闪存设备会提供读和写状态寄存器的指令，包括 write-in-progress 比特掩码。

图 38 为记录 RDSR 和 WRSR 命令的数据单 M25910-A 的一部分。图 43 中的图示出了 WIP 比特在 SR 的比特位置 0，导致比特掩码为 0x01。

28.13 添加支持到 xTimeComposer

可以与 libflash 或 xflash 一起使用配置文件。下面的实例程序使用 libflash 连接到 M25P10-A 设备上，配置参数规定在 m25p10a 中。

```
#include "platform.h"
#include "flash.h"
#include "flashlib.h"
#include "stdio.h"
#include "stdlib.h"

fl_PortHolderStruct SPI = {PORT_SPI_MISO,
                           PORT_SPI_SS,
                           PORT_SPI_CLK,
                           PORT_SPI_MOSI,
                           XS1_CLKBLK_1};

fl_DeviceSpec myFlashDevices[] = {
 {
   #include "m25p10a"
 }
};

int flash_access() {
  if (fl_connectToDevice(SPI, myFlashDevices,
      sizeof(myFlashDevices)/sizeof(fl_DeviceSpec)) != 0) {
    printf("No supported flash devices found.\n"); exit(1);
  } else {
  printf("Found custom flash device m25p10a.\n"); exit(0);
  }
  return 0;
}

int main() {
  // multicore main is required for xscope
  par {
    on stdcore[0] : flash_access();
  }
}
```

必须在 XN 文件中规定自定义闪存设备，具体如下：

```
<ExternalDevices>
  <Device NodeId="0" Tile="0" Name="bootFlash"
          Class="SPIFlash" Type="M25P10A">
    <Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO" />
    <Attribute Name="PORT_SPI_SS" Value="PORT_SPI_SS" />
    <Attribute Name="PORT_SPI_CLK" Value="PORT_SPI_CLK" />
    <Attribute Name="PORT_SPI_MOSI" Value="PORT_SPI_MOSI" />
  </Device>
</ExternalDevices>
```

为了汇编链接到 lib 闪存库的图像文件，启动命令行工具（参见§3.2）并输入以下命令：

```
▶ xcc main.xc -o prog.xe -target=target_with_custom_flash.xn -lflash
```

输入以下命令，生成可以在之后编程到上述闪存设备中的 xCORE 闪存格式的图像文件：

```
▶ xflash prog.xe -o imgfile --spi-spec m25p10a
```

XFLASH 生成用于自定义闪存设备的图像，并将之写入规定的图像文件中。

28.14 选择闪存设备

在选择与 xCORE 设备一起使用的闪存设备时，建议遵循以下指南：

· 如果需要访问数据部分，则应选择一个具有细粒度消除粒度的设备，因为这样会将工厂和升级图像之间的间隙最小化，还可以将在写数据时 libflash 需要缓冲的数量最小化。

· 如果可能，选择具有矢量保护的设备以确保引导装载程序和工厂图像有保护，防止在部署后意外受损。

· 选择一个适用于应用的闪存速度等级。即便在低下启动时间也是最短的。

XMOS

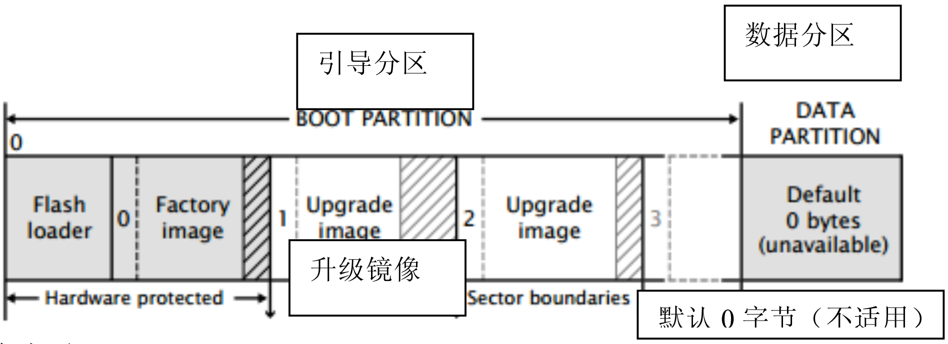
29 XFLASH 命令行手册

本章内容

- •总体选项
- •目的选项
- •安全选项
- •编程选项

如下图所示，XFLASH 可用于创建 XCORE 闪存格式的二进制数。该命令还可以将此类文件用于启动 XMOS 系统的闪存设备。

图 64：Flash 格式图



29.1 总体选项

下列选项用于指定程序的镜像和数据所构成的二进制数字及其布局。在插入填充时必须确保镜像对齐扇区边界。

xe-file [size]

--factory xe-file [size]

指定的 Xe 文件为原厂的镜像。如果相应尺寸被指定，填充的插入使这一镜像的开始和下一个镜像之间的空间至少为指定的容量。尺寸默认单位是“字节”，可以后缀 K 来指定单元的字节。

可指定最多一个原厂镜像。

--upgrade id xe-file [size]

指定的文件版本 ID 为 Xe 各版本号升级镜像必须是一个大于 0 的独一无二的号码。如果容量被指定，填充的是插入使这一镜像的开始和下一个镜像之间的空间至少为指定的容量。尺寸默认单位是“字节”，容量可以后缀 K 来指定单元的字节。

多个升级镜像按照命令行中指定的顺序插入启动分区。

如未指定任何原厂镜像，单一的升级镜像可以被指定和写入选项文件。

--boot-partition-size n

指定的启动分区的容量为 n 个字节。如果未指定，使用默认容量的闪存设备的总容量。n 必须大于或等于所需的存储引导程序、原厂镜像和任何升级镜像的最小尺寸。

--data file 指定要写入的数据分区的文件的内容。

--loader file 指定自定义闪存装载函数文件（见§22.4）。文件可以是一个对象（.O）或档案（.a）。

默认情况下，该 xcore Flash 用最高版本加载镜像，验证其 CRC。

--verbose 印刷品附加信息的程序加载到目的系统。

--help 打印一个描述支持指令的选项。

--version 显示版本号和版权。

29.2 目的选项

下列选项用于指定哪个闪存装置的二进制为可编程的。闪存装置使用的类型决定了 SPI 分频器值，扇区的容量和内存容量。

--list-devices

-l 打印所有连接到 PC 的 JTAG 适配器的枚举列表和每个 JTAG 链中的设备，形式如下：

ID 名称 适配器 ID 设备

-- ----

适配器按其序列号订购。

--id ID 指定连接到目的硬件的适配器。

XFlash 连接到目的平台并确定连接到它的闪存装置的类型。

--adapter-id ADAPTER-SERIAL-NUMBER

指定连接到目的硬件的适配器序列号。

XFlash 连接到目的硬件并确定连接到它的闪存装置的类型。

XMOS

--jtag-speed n

设置用于 JTAG 时钟的分频器。相应的 JTAG 时钟速度为 6/(n+1)兆赫。分压器的默认值为 JTAG 时钟是 0，代表 6 兆赫。

--spi-spec file

指定的可支持的闪存设备见文件（见§25）。

--spi-div n 设置用于 SPI 时钟的分频器，生产 100 / 2n 兆赫的 SPI 时钟速度。默认情况下，如果未指定目的，分频器的值设置为 3（16.7 兆赫）。

--noinq 不运行驱动程序的检查设备，核对检查镜像对齐扇区边界。如果 –noinq 被省略，期望 XFlash 能够通过 JTAG 连接到设备。

--disable-boot-link-warn

禁用警告发出时，节点之间的联系不允许从链接到工作的启动，例如仅低于已被指定为连接节点的链接 3，则引导 ROM 链接 4-7（如果链接 3 为主瓦）。

29.3 安全选项

下列选项应与 AES 模块一同使用（见第 27.1 节）。

--key keyfile

用密匙文件加密引导分区中的镜像。.

--disable-otp

使闪存装载在程序启动时禁止访问 OTP 存储器。这是默认的选项——加密。

--enable-otp

使闪存装载在程序启动时能够访问 OTP 存储器。这是默认的选项——加密

29.4 编程选项

默认情况下，XFlash 程序生成二进制文件到目的的闪存设备。

-o file 位置输出文件，禁用程序。

如果目的平台是从一个以上的闪存装置启动，为每一个装置创建多个输出文件。每个输出文件的名称是 file_node，在相应节点的 ID 属性的值（见§42.4）。

下列选项在目的闪存设备上执行通用的读取、写入和擦除的操作。目的的文件必须被指定，它提供了用于对硬件平台的 SPI 设备通信端口。

--target-file xn-file [node]

XMOS

指定的 xn-file 文件作为目的平台。

如 xn-file 指定多个闪存装置，则节点的值必须被指定。该值必须对应连接到目的闪存设备节点的 ID 属性（见§42.4）。

--target platform [node]

指定一个目的平台。该平台的配置必须在文件 platform.xn 被指定，被搜索的路径由 xcc_device_path 环境变量指定（见§9.8）。

如果 xn-file 指定多个闪存装置，则节点的值必须被指定。该值必须对应连接到目的闪存设备节点的 ID 属性（见§42.4）。

--erase-all 清除闪存装置的所有存储。

--read-all 读取闪存设备的所有内存内容并写入到主机。必须同-o 共同使用。

--write-all file

以字节为单位的文件写入到闪存设备。

XMOS

第 K 部分
安全和一次性编程

目录

30 维护 IP 和设备准确度

本章内容

•xcore AES 模块

• AES 开发模块的启用

•生产闪存编程流程

•生产 OTP 编程流程

Xcore 设备包含一次性可编程（OTP）的片上存储器，内存可在装置测试期间或之后展开。可将 xcore AES 模块分散到 OTP 设备，允许程序被加密存储在闪存设备。这有助于提供：

- 保密性

加密程序难以被反向编程。

- 程序真实性

AES 装载机将无法加载被篡改的程序或其他第三方程序。

- 设备的真实性

无法使用由第三方提供的 XCore 设备复制由您的密钥加密的程序。

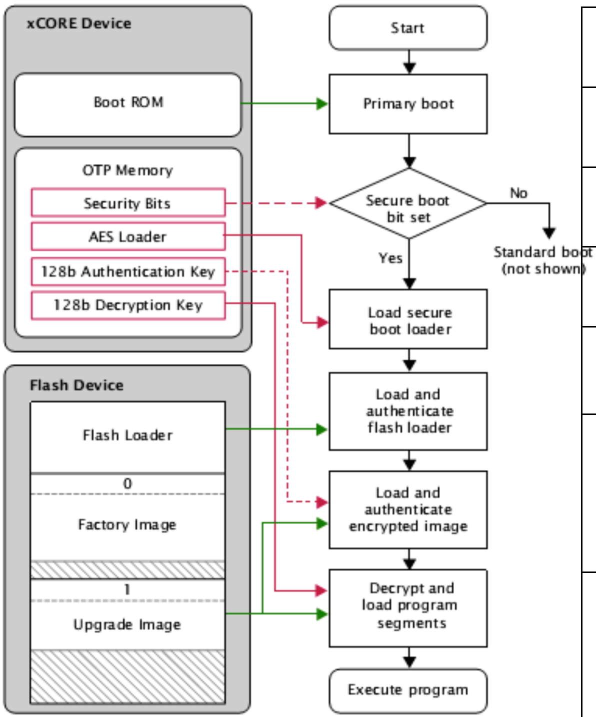
一旦 AES 模块编程完成，OTP 安全位将被断开，每个区块生成一个“安全孤岛”所有计算、内存、I / O 和通信将接受相应区块上运行代码的单独控制。在设置时，此类位置：

- 将强制通过 OTP 引导以防止被绕过，
- 禁用 JTAG 访问区块以防止按键被读取，并停止任何其他 OTP 写入以防止更新。

AES 模块针对普通黑客提供了强有力的安全保护，因此很有必要实现。但重要的是要意识到，不存在绝对牢不可破的安全性，如果对方决心已定且具备所需能力，任何方法都无法阻止对方获得您的密匙。

30.1 XCore AES 模块

XCore 编辑 AES 模块验证并解密 SPI 闪存设备中的程序。如图 47 所示，将其编程至设备后可启用以下安全开机程序。



xCORE 设备	开始
Boot ROM	首次启动
OTP 内存	安全启动位设置 否 标准启动（未显示） 加载安全启动器
安全位	
AES 加载器	
128 位认证密钥 128 位密钥	
闪存设备	加载和认证 闪存加载器
闪存加载器	加载和认证加密镜像
0 原厂镜像	解密并加载程序段。
1 升级镜像	执行程序

图 65:

使用 AES 模块的

安全开机程序

1. 该装置主要由 ROM 引导程序加载，所检测到的安全启动位设置在 OTP 然后加载并执行 AES 模块。

2. AES 模块加载内存至 SPI Flash 装载机

3. AES 模块使用 cmac-aes-128 算法和 128 位密钥验证闪存装载机。如果认证失败，启动停止。

4. AES 模块将认证密钥和解密密钥存放于寄存器并跳转到闪存装载机。

闪存装载机执行下列操作：

1. 选择验证其 CRC 数字最大的镜像
2. 对选定的镜像使用 CMAC 标签和认证密钥。如果认证失败，启动停止。
3. 验证、解密和负载程序/数据段到内存中的表。如果镜像没有认证，启动停止。
4. 开始执行程序。

多节点系统，AES 模块写入某区块的 OTP，和从 xConnect 链路协议安全启动编程到所有其他区块。

30.2 启用 AES 开发模块

您可以在开发或设备制造过程的任何时间激活 AES 模块。在开发环境中，您可以离开没有设置的安全位并激活模块，从而使：

- XFlash 使用设备程序加载到闪存，
- xgdb 调试程序上运行的装置以及
- Xburn 编写额外的 OTP 位保护装置。

在生产环境中必须保护装置，防止 OTP 按键被最终用户读出。

如需将 AES 模块编程至开发板中的 XCore 设备，启动命令行工具（参见第 3.2 节）并输入以下命令：

1. xburn --genkey keyfile

XBURN 将两行随机的 128 位密钥写入 keyfile。第一行为上述认证密钥，第二行为解密密钥。

可使用开放源码库 crypto++生成加密密钥。如愿意，您也可创建该文件并提供自有钥匙。

2. xburn -l

XBURN 打印所列所有连接至您的电脑的 JTAG 适配器列表以及 JTAG 链中的每个装置，相应形式为：

ID - NAME (ADAPTER-SERIAL-NUMBER)

3. xburn --id ID --lock keyfile --target-file target.xn --enable-jtag --disable-master-lock

XBURN 将 AES 模块和安全密钥写入连接至目的设备的 OTP 存储器并设定其安全开机位。用于引导的 SPI 端口将从 XN 文件中引出（见 XM-000929-PC）。

XMOS

如需加密程序并将其写入闪存，请输入命令： • xflash --id ID bin.xe --key keyfile

要保护的 XCore 设备并防止任何进一步开发，请输入以下命令：

- xburn --id ID --target-file target.xn --disable-jtag --lock keyfile

30.3 生产闪存编程流程

在生产制造环境下，相同的程序通常被编程至多个 SPI 设备。

要生成一个加密 XCore 闪存格式的镜像，启动命令行工具（见§3.2）并输入以下命令：

- xflash prog.xe -key keyfile -o image-file

使用第三方闪存编程可将此镜像直接写入闪存，也可使用 XFLASH 将其写入：(通过 XCore 编辑设备)。如需使用 XFLASH 写入，请输入以下命令：

1. xflash -l

XFLASH 打印所列所有连接至您的电脑的 JTAG 适配器的枚举列表以及 JTAG 链中的每个装置，相应形式为：

ID - NAME (ADAPTER-SERIAL-NUMBER)

2. xflash --id ID --target-file platform.xn --write-all image-file

XFLASH 生成 XCore 闪存格式的镜像，其中包含第一阶段的加载器和原厂镜像以及来自您编译程序的二进制和数据段。随后 XFLASH 将使用 XCore 设备写入该镜像至闪存。

XN 文件必须定义一个 SPI 闪存设备以及其连接的 XCore 设备的四个指定端口（见 XM-000929-PC）。

XMOS

30.4 生产 OTP 编程流程

在生产制造环境下，相同的密匙通常被编程至多个 XCore 设备。

如需生成包含 AES 模块和安全密钥并将被写入 OTP 的镜像，启动命令行工具（见第 3.2 节）并输入以下命令：

1. xburn --genkey keyfile

XBURN 将两行随机的 128 位密钥写入 keyfile。第一行为上述认证密钥，第二行为解密密钥。

可使用开放源码库 crypto++生成加密密钥。如愿意，您也可创建该文件并提供自有钥匙。

2. xburn --target-file target.xn --lock keyfile -o aes-image.otp

XBURN 生成包含 AES 模块、安全密钥的安全位取值的镜像。

该镜像包含密匙，不得泄密。

如需在生产环境中将 AES 模块和安全位写入设备，输入以下命令：

1. xburn -l

XBURN 打印所列所有连接至主机的 JTAG 适配器枚举列表以及 JTAG 链中的每个装置，相应形式为：

ID - NAME (ADAPTER-SERIAL-NUMBER)

2. xburn --id ID --target-file target.xn aes-image.otp

XBURN 在设备中加载将 AES 模块和安全密钥写入 OTP 的程序并设定其安全开机位。用于引导的 SPI 端口将从 XN 文件中引出（见 XM-000929-PC）。XBURN 返回 0 为成功，非零为失败。

31 XBURN 命令行手册

本章内容

- •整体选项
- •安全寄存器选项
- •目的选项
- •编程选项

XBURN 创建 OTP 镜像并将镜像编入 XCore 编辑设备的 OTP 存储器之中。

31.1 整体选项

下列选项被用于指定 OTP 镜像和安全寄存器的内容。

xe-file 指定通过 XE-文件和一组默认的安全位在装入段构造的开机镜像（见图 48）。

otp-file 指定 OTP 文件内的 OTP 分段，其中需包括安全寄存器的取值。

--lock keyfile

指定 XCore 编辑 AES 启动模块（见第 27.1 节）和默认安全位的集合（见图 48）。

--genkey keyfile

将两个用于认证和加密的 128 位密钥输出至 keyfile。使用开放源码库 crypto++生成密钥。

在--burn or --lock 下此选项是无效的

--mac-address mac

MAC 地址写入 OTP 的结尾。MAC 地址应该以 12:34:56:78:9 A：BC 的形式指定。可以通过多次指定 - MAC 地址选项写入多个 MAC 地址。 MAC 地址写入至 OTP 的顺序与选项出现的顺序相同。

--serial-number 序列号

将某个 32 位的序列号写入 OTP 的结尾。

--read 打印 OTP 的全部内容。

--help 打印所支持的命令行选项的描述。

--version 显示版本号和版权。

XMOS

31.2 安全寄存器选项

下列选项用于指定 OTP 安全寄存器的内容并覆盖图 48 所示 BURN XE 镜像、OTP 镜像和 AES 模块的默认选项。

图 48：由 XBURN 写入的默认安全位

安全位	XE 镜像	OTP 镜像	AES 模块 (--锁住)
OTP Boot	启用	对应每个 OTP 镜像文件	启用
JTAG Access	启用		禁用
Plink Access	启用		启用
Global Debug	启用		禁用
Master Lock	禁用		启用

--enable-otp-boot

允许从 OTP 启动。

--disable-jtag

禁用 JTAG 访问。一旦禁用，将不可再次获得设备调试访问权力或读取 OTP。

此选项不禁用边界扫描。

--disable-plink-access

禁止通过其他区块访问 plink 寄存器。禁用伪链接访问将限制所有通过每个 plink 本地区块对该 plink 寄存器的所有存取。

--disable-global-debug

防止设备加入全局调试。禁用全局调试将阻止区块使用全局调试引脚进入调试。

--enable-master-lock

启用 OTP 主锁。即不允许对 OTP 进行任何进一步修改。

31.3 目的选项

下列选项用于指定目的硬件平台。

--list-devices

-l 打印所列全部 JTAG 适配器列表，相应适配器应通过各个 JTAG 链按如下形式连接至主机和设备：

ID - NAME (ADAPTER-SERIAL-NUMBER) 应按照其序列号订购此类适配器。

--id ID 指定连接到目的硬件的适配器。

XMOS

--adapter-id ADAPTER-SERIAL-NUMBER

指定连接到目的硬件的适配器的序列号。

--jtag-speed n

设置 JTAG 时钟分频器为 n。相应 JTAG 时钟速度是 6 /（n +1）兆赫。默认值是 0（6MHz）。

--spi-div n 设置 AES 模块所用分压器的的 SPI 时钟至 n。相应的 SPI 时钟速度被设置为 100 /（2n）兆赫。默认值为 20（2.5MHz）。

在--lock 下此选项才为有效。

--target-file xn-file

指定 XN-文件为目的平台。

--target platform

指定一处目的平台。在文件 platform.xn 中必须指定该平台的配置，可通过由 XCC_DEVICE_PATH 环境变量（参见第 9.8 节）中指定的路径搜索该本件。

31.4 编程选项

默认情况下，XBURN 将指定的 OTP 镜像写入目的平台。 -o otp-file 将输出至 otp-file, 从而禁用编程。

--make-exec xe-file

将一个可执行文件放入 xe-file，于 XCore 编辑设备运行时执行规定的 OTP 烧录操作并禁用编程。

也可随后使用 XRUN 运行该 XE 文件。

--force

-f 禁止在提示后写入 OTP。此并非默认值。

--size-limit n

限制写入 OTP 前 n 个字节的 OTP 存储器数量。如果镜像不符合规定的限制将出现一次错误。

XMOS

第 L 部分

C/ XC 语言编程

目录

32 C / C ++和 XC 之间的调用

本章内容

•XC 至 C/ C + +的参数传递

•C / C++至 XC 参数传递

在某些情况下，可将 XC 语言的某类型参数转移至 C / C ++语言的非同类型函数参数，反之亦然。

为简化 C / C ++和 XC 之间常用函数的说明任务，系统头文件 xccompat.h 的包含多种类型的定义和宏定义。请参阅头文件相关文档。

32.1 XC 至 C/ C + +的参数传递

带有 unsigned int 类型的参数 C / C++ 语言定义函数可使用 XC 语言阐明，相应参数的类型可为端口、通道传输结束或计时器。

带有“指向 T”类型参数 C / C++ 语言定义函数可使用 XC 语言阐明，相应参数的类型为“参考 T”或“可为空的参考 T”。

带有“指向 T”类型参数 C / C++ 语言定义函数可使用 XC 语言阐明，相应参数的类型为“T 阵列”。

32.2 C / C++至 XC 参数传递

带有端口、通道传输结束或计时器类型参数 XC 语言定义函数可使用 C/C++语言阐明，相应参数的类型为 unsigned int。

带有“参考 T”或“可为空的参考 T”类型参数 XC 语言定义函数可使用 C/C++语言阐明，相应参数的类型为“指向 T”。

带有“T 阵列”类型参数 XC 语言定义函数可使用 C/C++语言阐明，相应参数的类型为“指向 T 类型”。在这种情况下，链接程序插入代码 XCore 编辑将添加一个等于无符号整数类型最大值的隐式数组边界参数。

33 XC 语言实现定义的行为

需要一致性 XC 应用才可记录其被指定执行定义的语言规范的所有部分的行为选项。在以下章节中，所有依赖于外部定义的应用程序二进制接口的选项将被列为“符合 ABI 定义”，且其记录应符合应用程序二进制接口规范（参见第 40 节）。

- 多字符常量取值（第 1.5.2 节）。

  多字符常量取值应与其首个字符取值相同，所有其他字符将被忽略。

- 相同字符串的文字是否不同对此没有影响（第 1.6 节）。

  如相同字符串的文字不同；其实现方式应存在单一的内存位置。

- 可被存储至某 char 的可用取值范围以及是否带有签名（第 3.2 节）。

  Char 的容量为 8 位。存储至 char 的变量值是否带有签名由 ABI 决定。

- 连接至某个端口并用于与环境进行通信的管脚数；不带外部说明其非显式初始化某个端口或时钟取值（第 3.2，7.7 节）。

  连接至某端口用于环境通信的针脚数目的定义由显式初始化程序或其说明符决定。如不存在初始化程序，编译器会生成一份错误消息。

- 某端口的理论传输类型、某端口的理论计数器类型以及某计时器的理论计数器类型（第 3.2 节）。

  ABI 将决定理论类型。

- 某整数值转换为带符号类型的情况应为其取值无法使用新类型表示（第 5.2 节）。

  当任何整数值转换为带符号类型且其取值无法使用新类型表示时，其数值将被截短至新类型的宽度和符号扩展。

•溢出处理、除法校验和其他异常情况的表达式评估（第 6 节）。

导致未定义的行为的所有条件（零除、零模组以及符号除法/模组溢出）。

- 对齐（第 6.3.4 节）的概念。

  2n 的校准可保证有效 n 位内存地址为零。上述特定校准类型由 ABI 确定。

- 运算符结果的取值和类型（第 6.4.6 节）。

运算符操作符结果的取值由 ABI 决定。结果类型为不带符号的 int。

•试运行时间的零除数（第 6.6 节）。

试运行时间的零除数将导致自陷

- 使用 inline 函数说明符创建暗示的程度应为有效的（第 7.3 节）。

inline 函数说明符被作为内联函数的某个提示。编译器将尝试在所有-O0 以上的优化级别内内联函数。

- 使用寄存器存储类说明符创建暗示的程度应为有效的（第 7.7.4 节）

寄存器存储说明符可使寄存器分配者尝试将变量置于函数中的寄存器内。但分配者并非一定将其置于某个寄存器之中。

- 支持输入操作所谓词函数（第 8.3 节）。

所支持的谓词函数集被记录于第 39.8 节中

- 输入和输出端口（第 8.3.2 节）的涵义。

输入和输出端口映射的是输入和输出端口的资源指令，其中的行为定义可参见 XS1 接口规范（见 X1373）。

- 达到底层通信协议根据交易通信需要优化的程度（第 8.9 节）。

通信协议由 ABI 决定。

- 无论是否由于通信发生后，相应输出的字节数不等于输入的子节数以及通信的取值而导致交易无效（第 11 节）。

所有情况下皆由 ABI 决定。

- 无效操作的行为（第 12 节）。

除下文所述以外，所有无效操作应被报告为编译错误或将在运行时导致自陷。

- 无效主交易说明的行为并不确定；无效从属交易将始终自陷。
- 不安全编译指令（见第 8 节）可以用来禁用特定的安全检查，这将导致无效操作的未定义行为。

XMOS

34 C 语言实现定义的行为

本章内容

需要一致性 C99 应用才可记录其被指定执行定义的语言规范的所有部分的行为选项。除如下所列选项以外，xTIMEcomposer 实现定义的行为应与 GCC4.2.1[5]相匹配。

下一章节的标题是指 C99 规范中的相应部分（见第 32.1 节），所有依赖于外部定义的应用程序二进制接口的选项将被列为“符合 ABI 定义”，且其记录应符合应用程序二进制接口规范（参见第 40 节）。

只有被支持的 C99 函数有相关记录。

34.1 环境

•当程序在一个独立的环境启动时所调用函数的名称和类型（5.1.2.1）。

已提供某种托管环境。

•其中主要函数可以被定义的另一种方式（5.1.2.2.1）。

没有可以定义主要函数的替代方式。

• 字符串的给定值由 argv 参数指向主要函数（5.1.2.2.1）。

argc 值等于零。argv[0]是一个零指令字。不存在其他数组成员。

•某个互动装置的构成部分（5.1.2.3）。

所有数据流皆适用于交互式设备。

• 对应计算异常其除 SIGFPE、SIGILL 和 SIGSEGV 之外的其他信号值（7.14.1.1）。

[5] http://www.xmos.com/references/gcc-4.2.1-c-implementation

不存在其他对应计算异常的信号值。

- 相当于该信号（sig，SIG_IGN）的信号值应在程序启动执行（7.14.1.1）。

在程序启动时将为所有信号执行信号（SIG，SIG_DFL）的等价值。

- 环境名称的组合以及更改环境列表中所使用 getenv 函数（7.20.4.5）的方法。

环境名称的组合为空。不存在可改变 getenv 函数所使用环境列表的方法。

- getenv 函数所使用的系统函数执行字符串的方式（7.20.4.6）。

相应情况由执行环境决定。

34.2 标识符

•单个标识符内重要的初始字符数（5.2.4.1，6.4.1）。

标识符中的所有字符（带或不带外部联动）皆为重要的。

34.3 字符

- 执行字符集（5.2.1）组成部分的取值。

应由 ASCII 字符集决定。

- 为各标准字母转义序列而产生的执行字符集组成部分的独特取值（5.2.2）。

应由 ASCII 字符集决定。

- 某个字符对象的值，其中已经存储了任何不为基本执行集组成部分的字符（6.2.5）。

任何不为基本执行集组成部分的字符的值由 ASCII 字符集决定。

•源代码字符集组成部分（字符常量和字符串文字）对执行字符集（6.4.4.4，5.1.1.2）组成部分的映射。

源代码字符集必须使用 ASCII 字符集。源代码字符集内部每个字符都将被映射至执行字符集内的相同字符。

- 包含多个字符或包含一个并未映射至某单字节执行字符的字符或转义序列的字符常量的取值。

包含多个字符的字符常量的取值等于该字符常量内最后一个字符的取值。包含一个并未映射至某单字节执行字符的字符或转义序列的字符常量的取值等于减去 modulo 2n 的取值，以达到相应的字符类型范围，其中 n 为单个字符的字节数。

•含有一个以上的多字节字符或包含多字节字符或转义的多字节字符又或包含扩展执行字符集并未表明的多字节字符或转义序列的宽字符常量的值（6.4.4.4）。

宽字符常量不能包含多字节字符。

• 用于转换宽字符常量的当前语言环境，该常量所包含的某单一多字节字符将扩展执行字符集的组成部分映射至对应的宽字符代码（6.4.4.4）。

宽字符常量不能包含多字节字符。

• 包含扩展执行字符集（6.4.5）并未表明的多字节字符或转义序列的某个字符串常量的值。

字符串字面值不可包含多字节字符。如果某个字符串字面值中使用了非执行字符集表达的转义序列，字符串中相应字符的值应等同于将会赋予作为转义序列组成部分的字符常量的值。

34.4 浮点运算

• 浮点运算以及返回<math.h>和<complex.h>浮点结果（5.2.4.2.2）的库函数的准确性。

故意未保存相关记录。

• 其他浮点异常、舍入模式、环境和分类的宏名称（7.6，7.12）。

没有其他被定义的浮点异常、舍入模式、环境或分类。

34.5 提示

▼ 使用 register 存储类说明符提出建议的程度是有效的（6.7.1）。

除非用作 register 变量扩展的一部分，否则忽略 register 说明符。

34.6 预处理指令

▼ 每个已识别的非 STDC #pragma 指令的行为（6.10.6）。

记录在 ξ8 中。

34.7 库函数

▼ 独立式程序可用的任何库工具，子句 4 要求的最低设置除外（5.1.2.1）。

提供主机环境。

XMOS

▼ 断言宏输出的诊断格式（7.2.1.1）。

断言宏采用的格式为："断言失败：expression，文件 filename、行 line number、函数：function。"，其中 expression 是指参数文本，filename 是_ _FILE_ _的值，line number 是_ _LINE_ _的值，而 function 则是流函数的名称。如果无法确定流函数的名称，则省略消息的该部分。

▼ 由 fegetexceptflag 函数存储的浮点状态标志的表示（7.6.2.2）。

不支持函数 fegetexceptflag。

▼ 除了"上溢"或"下溢"浮点异常之外，feraiseexcept 函数是否会产生"不精确"浮点异常（7.6.2.3）。

不支持函数 fegetexceptflag。

▼ 除了"C"和""之外可以作为第二个参数传递给 setlocale 函数的其他字符串（7.11.1.1）。

其他字符串不可作为第二个参数传递给 setlocale 函数。

▼ 当 FLT_EVAL_METHOD 宏的值小于 0 或大于 2 时针对 float_t 和 double_t 定义的类型（7.12）。

不支持 FLT_EVAL_METHOD 宏的其他值。

▼ 数学函数的域错误，与此国际标准的要求不同（7.12.1）。

故意留白。

▼ 发生下溢范围错误时数学函数返回的值，当整数表达式 math_errhandling & MATH_ERRNO 为非零时，是否将 errno 设置为宏 ERANGE 的值，以及当整数表达式 math_errhandling & MATH_ERRNO 为非零时，是否产生"下溢"浮点异常。（7.12.1）。

故意留白。

▼ 当一个 fmod 函数具有一个等于零的第二参数时，是发生域错误还是返回零（7.12.10.1）。

当一个 fmod 函数具有一个等于零的第二参数时，发生域错误。

▼ 在减小商时，remquo 函数所用的以 2 为基数的模数对数（7.12.10.3）。

商是减小后的模数 27。

▼ 在调用信号处理程序之前是否执行 signal(sig, SIG_DFL); 的等效函数，如果不执行，则阻塞执行的信号（7.14.1.1）。

在调用信号处理程序之前执行 signal(sig, SIG_DFL); 的等效函数。

XMOS

▼ 宏 NULL 扩展到的空指针常量（7.17）。

确定 NULL 为（(void *）0）。

▼ 文本流的最后一行是否需要一个终止换行符（7.19.2）。

由执行环境确定。

▼ 写出到换行符前面的文本流中的空格字符在读入时是否出现（7.19.2）。

由执行环境确定。

▼ 可附加至写入二进制流的数据的空字符数（7.19.2）。

由执行环境确定。

▼ 附加模式流的文件位置指令符最初是位于文件开头还是文件结尾（7.19.3）。

由执行环境确定。

▼ 对文本流的写操作是否会导致关联文件在超过该点后被截断（7.19.3）。

由执行环境确定。

▼ 文件缓冲的特征（7.19.3）。

缓冲的输出流保存字符，直至缓冲满了为止，然后将所述字符作为块进行写入。行缓冲的输出流保存字符，直至行完成或缓冲满了为止，然后将所述字符作为块进行写入。未缓冲的输出流立即将字符写入目的文件。

▼ 零长度文件是否确实存在（7.19.3）。

由执行环境确定。

▼ 书写有效文件名的规则（7.19.3）。

由执行环境确定。

▼ 同一文件是否可以同时打开多次（7.19.3）。

由执行环境确定。

▼ 文件中多字节字符所用编码的性质和选项（7.19.3）。

执行字符集不得包含多字节字符。

▼ remove 函数对打开的文件的作用（7.19.4.1）。

由执行环境确定。

▼ 在调用 rename 函数之前存在一个具有新名称的文件时的作用（7.19.4.2）。

XMOS

由执行环境确定。

▼ 程序异常终止后是否删除打开的临时文件（7.19.4.3）。

程序异常终止后，不删除临时文件。

▼ 允许模式进行哪些更改（如果有），以及在什么情况下允许更改（7.19.5.4）。

不得赋予文件更自由的存取模式（比如，在只读文件描述符上，“w”模式将失效），但可以改变状态如附加或二进制模式等。如果不能更改，将发生故障。

▼ 用于输出无穷大或 NaN 的样式，以及为 NaN 输出的任何 n-char 或 n-wchar 序列的含义（7.19.6.1，7.24.2.1）。

代表无穷大的双参数在样式[-] inf 中转换。代表 NaN 的双参数在 as nan 样式中转换。

▼ 在 fprintf 或 fwprintf 函数中%p 转换的输出（7.19.6.1，7.24.2.1）。

指针的值被转换为 dddd 样式的无符号十六进制计数；利用字母 abcdef 进行转换。精度规定出现的最少数字位数；如果被转换的值可以用较少数字表示，则用前导零加以扩展。默认精度为 1。将字符 0x 添加到输出之前。

不支持 fwprintf 函数。

▼ 在 fscanf 或 fwscanf 函数中 %[转换的扫描列表中，当 - 字符既不是第一个字符也不是最后一个字符，又不是 ^ 字符为第一个字符的情况下的第二个字符时，此时对 - 字符的解释（7.19.6.2，7.24.2.1）。

如果在 – 字符之后的字符在数值上比在 – 字符之前的字符大，则认为 – 字符限定了一个范围。否则 – 字符本身应被加入扫描列表。

不支持 fwscanf 函数。

▼ fscanf 或 fwscanf 函数中%p 转换匹配的序列集合以及相应输入项的解释（7.19.6.2，7.24.2.2）。

%p 与%x 匹配同样的格式。相应的输入项被转换为一个指针。不支持 fwscanf 函数。

▼ 在表示由 strtod、strtof、strtold、wcstod、wcstof 或 wcstold 函数转换的 NaN 的字符串中的任一 n-char 或 n-wchar 序列的含义(7.20.1.3，7.24.4.1.1)。

不支持 wcstod、wcstof 和 wcstold 函数。在表示 NaN 的字符串中的一个 n-char 序列应以十六进制的形式扫描。忽略所有非十六进制数位的字符。

▼ 发生下溢时，strtod、strtof、strtold、wcstod、wcstof 或 wcstold 函数是否将 errno 设置为 ERANGE（7.20.1.3，7.24.4.1.1）。

不支持 wcstod、wcstof 和 wcstold 函数。发生下溢时，函数 strtod、strtof 和 strotold 不会将 errno 设置为 ERANGE 并归零。

XMOS

▼ 当请求的大小为零时，calloc、malloc 和 realloc 函数是返回空指针还是返回指针至一个已分配对象（7.20.3）。

当请求的大小为零时，函数 calloc、malloc 和 realloc 均返回空指针至一个已分配对象。

▼ 调用 abort 或_Exit 函数时，是刷新具有未写入缓冲数据的开放流、关闭开放流还是删除临时文件（7.20.4.1，7.20.4.3，7.20.4.4）。

调用 abort 函数或_Exit 函数时，删除临时文件，不刷新缓冲文件，也不关闭开放流。

▼ abort、exit 或_Exit 函数返回给主机环境的终止状态（7.20.3）。

函数 abort 导致软件异常发生。函数 exit 和_Exit 返回给主机环境的中止状态由执行环境确定。

▼ system 函数在其参数不是空指针时返回的值（7.20.4.6）。

由执行环境确定。

▼ 可以用 clock_t 和 time_t 表示的时间的范围和精度（7.23.1）。

用 time_t 表示的时间精度由执行环境界定。time_t 指定一个无符号长整数。用 time_t 表示的实际时间范围由执行环境界定。

宏 CLOCKS_PER_SEC 被界定为 1000。clock_t 指定一个无符号长整数。

▼ clock 函数的年代（7.23.2.1）。

clock 函数总是返回值（clock_t）(-1)，表明所使用的处理机时间不可用。

▼ “C”语言环境中 strftime 和 wcsftime 函数的%Z 说明符的替换字符串（7.23.3.5，7.24.5.1）。

%Z 说明符用字符串“GMT”替换。

34.8 语言环境特定的行为

▼ 基本字符集之外的源代码字符集和执行字符集的附加成员（5.2.1）。

源代码字符集和执行字符集均包括 ASCII 字符集的所有成员。

▼ 基本字符集之外的执行字符集中附加多字节字符的存在、含义和表示（5.2.1.2）。

执行字符集中不存在多字节字符。

▼ 用于多字节字符编码的移位状态（5.2.1.2）。

源代码字符集和执行字符集中均不存在多字节字符。

▼ 连续输出字符的写入方向（5.2.2）。

字符从左到右输出。

▼ 小数点字符（7.1.1）。

小数点字符为‘.’。

▼ 输出字符集（7.4，7.25.2）。

由 ASCII 字符集确定。

▼ 控制字符集（7.4，7.25.2）。

由 ASCII 字符集确定。

▼ 由 isalpha、isblank、islower、ispunct、isspace、isupper、iswalpha、iswblank、iswlower、iswpunct、iswspace 或 iswupper 函数测试的字符集（7.4.1.2、7.4.1.3、7.4.1.7、7.4.1.9、7.4.1.10、7.4.1.11、7.25.2.1.2、7.25.2.1.3、7.25.2.1.7、7.25.2.1.9、7.25.2.1.10、7.25.2.1.11）。

不支持函数 isblank、iswalpha、iswblank、iswlower、iswpunct、iswspace 和 iswupper。

函数 islower 测试字符‘a’至‘z’。函数 isupper 测试字符‘A’至‘Z’。函数 isspace 测试字符‘‘，‘\f’，‘\n’，‘\r’，‘\t’和‘\v’。函数 isalpha 测试大写字符和小写字符。函数 ispunct 测试除空格字符和字母数字字符之外的所有可印刷字符。

▼ 本地环境（7.11.1.1）。

本地环境与 C 翻译的最小环境相同。

▼ 数值转换函数接受的附加主题序列（7.20.1，7.24.4.1）。

数值转换函数不接受任何附加主题序列。

▼ 执行字符集的整理序列（7.21.4.3，7.24.4.4.2）。

函数 strcoll 执行的比较与函数 strcmp 执行的比较相同。

▼ strerror 函数设置的错误消息字符串的内容（7.21.4.3，7.24.4.4.2）。

错误消息字符串的内容见图 49。

▼ iswctype 函数支持的字符分类（7.25.1）。

iswctype 支持的字符分类见图 50。

XMOS

图 67：错误信息字符串

值	字符串
EPERM	非所有人
ENOENT	没有这样的文件或目录
EINTR	中断系统调用
EIO	输入/输出故障
ENXIO	没有这样的设备或地址
EBADF	文件号不符合要求
EAGAIN	没有更多过程
ENOMEM	没有足够空间
EACCES	拒绝访问
EFAULT	错误的地址
EBUSY	设备或资源忙
EEXIST	文件存在
EXDEV	文件存在交叉链接
ENODEV	没有这样的设备
ENOTDIR	不是一个目录
EISDIR	是一个目录
EINVAL	非法参数
ENFILE	系统打开的文件太多
EMFILE	打开的文件太多
ETXTBSY	文本文件占用
EFBIG	文件过大
ENOSPC	设备空间不足
ESPIPE	非法查询
EROFS	只读文件系统
EMLINK	链接过多
EPIPE	管破裂
EDOM	数学参数
ERANGE	结果太大
ENAMETOOLONG	文件或路径名过长
ENOSYS	函数未执行
ENOTEMPTY	目录非空
ELOOP	符号链接过多

图 68：宽字符映射

值	描述
WCT_TOLOWER	转换为小写
WCT_TOUPPER	转换为大写

35 C 和 C++语言参考

本章内容

- ▼ 标准
- ▼ 书籍
- ▼ 在线

由于已存在高质量的文件，因此 XMOS 不制备与 C 和 C++标准语言特征相关的文件。

35.1 标准

- ▼ ISO/IEC 9899:1989：编程语言 — C.（C89）。国际标准组织。
- ▼ ISO/IEC 9899:1999：编程语言 — C.（C99）。国际标准组织。
- ▼ ISO/IEC 14882:2011：编程语言 — C++（C++标准）。国际标准组织。

35.2 书籍

▼ C 编程语言（第二版），Brian W. Kernighan 和 Dennis M. Ritchie 著，Prentice-Hall, Upper Saddle River, NJ, USA 于 1988 年出版，ISBN-10: 0131103628。

35.3 在线

▼ comp.lang.c 常见问题：http://c-faq.com/

XMOS

第 M 部分
用汇编语言编程

目录

36 内联汇编

可利用 asm 陈述将用汇编语言写成的编码嵌入一个 C 或 XC 函数中。比如，加法指令可以写成：

```
asm("add %0, %1, %2" : "=r"(result) : "r"(a), "r"(b));
```

用双点隔开汇编程序的模板、输出操作码和输入操作码。用逗号隔开一组中的操作码。用一个操作码约束字符串描述各操作码，后接用括号括起来的表达式。操作码约束字符串中的"r"表示该操作码必须位于一个寄存器中。操作码约束字符串中的"="表示该操作码是写成的。

各输出操作码表达式必须为一个左值，其约束中必须存在"="。

可利用格式%num（其中 num 表示操作码编号）的一个转义序列指出操作码在汇编程序中的位置。可利用转义序列"%="发出一个编号，该编号对一个 asm 陈述的各个表达式而言是唯一的。贴本地标签时这很有用。如需生成文字"%"，必须写入"%%"。

如果编码重写特定的寄存器，则用跟在输入操作码后面的第三个双点加以描述，后接 clobbered 寄存器的名称，作为一个由逗号分隔的字符串列表。比如：

```
asm ("getid r11\n\tmov %0, r11"
     : "=r"(result)
     : /* no inputs */
     : "r11");
```

编译器确保 clobbered 寄存器中无输入或输出操作码。

如果一个 asm 陈述具有输出操作码，则编译器假设该陈述除了写输出操作码外没有其它副作用。如果未使用 asm 陈述写的值，则编译器可删除该 asm 陈述。为了将一个 asm 陈述标记为具有副作用，应在 asm 后面加上关键字 volatile。比如：

```
asm volatile("in res[%1], %0" : "=r"(result) : "r"(lock));
```

如果 asm 陈述访问存储器，则在 clobber 寄存器列表中加入"memory"。比如：

```
asm volatile("stw %0, dp[0]"
     : /* no outputs */
     : "r"(value));
```

这能防止编译器将存储器值缓存至 asm 陈述周围的寄存器中。

可利用 earlyclobber 约束编译器"&"规定，在所有输入操作码被消耗完之前修改一个输

出操作码。这能防止编译器将该操作码与其它输入操作码放入同一寄存器中。比如：

```
asm("or %0, %1, %2\n"
    "or %0, %0, %3\n"
    : "=&r"(result)
    : "r"(a), "r"(b), "r"(c));
```

不支持从一个 asm 陈述转移到另一个 asm 陈述。不得将 asm 陈述用来修改任一资源的事件启用状态。

XMOS

37 编写与 XMOS XS1 ABI 兼容的汇编程序

本章内容

XMOS XS1 应用程序二进制接口（ABI）定义了用于由 C/C++、XC 和汇编码编译而来的对象的链接接口。本指南说明了如何写能够链接到 XMOS 编译器生成的对象的汇编码函数。

37.1 符号

由于汇编程序解析一个汇编文件，因此它能保持一个当前地址，每次当它分配存储区时，地址都会增加。

符号用来关联名称与地址。符号可在指令和指令中被引用，一旦值被计算出来，连接程序就会修补相应的地址。

以下程序定义了一个名为 f 的符号，指的是当前地址的值。它还让该符号全局可见，从其它文件也可查看，并且通过名称就能引用该符号。

```
# Give the symbol f the value of the current address.
f:
# Mark the symbol f as global.
  .globl f
```

通过写符号名称再加一点双点即可定义该符号。.globl 指令使符号从文件外部也可以看到。

37.2 对齐

XS1 ABI 规定了针对编码和数据的最低对齐要求。函数必须以对齐的 2 字节开头，数据必须逐字对齐。在一个符号的定义前加入.align 指令可使地址对齐。

以下程序定义了一个符号 f，确定其为下一个 2 字节对齐地址。

```
# Force 2 byte alignment of the next address.
  .align 2
f:
```

37.3 章节

各目的文件可能包含多个章节。当通过连接程序结合时，各目的文件中名称相同的章节将一起被放在连续的地址上。这样，在最终的可执行文件中，不同类型的编码或数据就可以分在一组。

XS1 ABI 要求将函数放在.text 章，将只读数据放在.cp.rodata 章，将可写数据放在.dp.data 章。默认章节为.text 章，可通过以下指令中的任意一个改变当前章节。

图 69：XMOS 连接程序支持的章节

章节	用于	指令
.text	可执行编码	.text
.dp.data	可写数据	.section .dp.data, “awd”, @progbits
.cp.data	只读数据	.section .cp.rodata, “ac”, @progbits

37.3.1 数据

以下示例程序定义了一个用值 5 初始化、在一个 4 字节边界对齐的 4 字节可写对象。

```
  .section .dp.data, "awd", @progbits
  .align 4
x:
  .word 5
```

你可以利用以下指令发出不同类型的数据。

指令	描述
.byte	发出一个 1 字节的值
.short	发出一个 2 字节的值
.word	发出一个 4 字节的值
.space	发出一个零初始化存储的 n 字节数组，其中 n 是指令参数
.asciiz	发出一个空结尾 ASCII 字符串
.ascii	发出一个 ASCII 字符串（无隐含的终止字符）

图 52：发出不同类型的数据的指令

37.3.2 数组

以下程序定义了一个大小为 42 字节的全局数组。

```
  .section .dp.data, "awd", @progbits
  .globl a
  .align 4
a:
  .space 42
  .globl a.globound
  .set a.globound, 42
```

XS1 ABI 要求每个全局数组 f 都具有一个相应的全局符号 f.globound，该符号用数组第一维度的元素数初始化。你可以利用.set 指令实施初始化。需要注意的是，如果该变量被一个 XC 函数使用，则将这个值用于数组界检查。

37.4 函数

XS1 ABI 规定了在函数之间传递参数和返回值的规则。此外，它还确定了用于规定函数所需硬件资源数量的符号。

37.4.1 参数和返回值

至多 32 位的标量值被当作 32 位值传递。最初的四个参数在寄存器 r0、r1、r2 和 r3 中传递，附加参数在堆栈中传递。同样，最初的四个返回值也在寄存器 r0、r1、r2 和 r3 中返回，附加值则在堆栈中返回。

在下列 XC 函数原型中，参数 a 和 b 在寄存器 r0 和 r1 中传递，返回值也一样。

```
{int, int} swap(int a, int b);
```

此函数的一个汇编实现如下所示：

```
  .globl swap
  .align 2
swap:
  mov r2, r0
  mov r0, r1
  mov r1, r2
  retsp 0
```


XMOS

37.4.2 调用者和被调用者保存寄存器

XS1 ABI 规定寄存器 r0、r1、r2、r3 和 r11 为调用者保存寄存器，其它所有寄存器则为被调用者保存寄存器。

调用一个函数之前，必须保存其值在调用完成后还需用到的所有调用者保存寄存器的内容。从一个函数返回后，所有被调用者保存寄存器的内容必须与其进入函数前的内容相同。

以下示例显示了一个采用被调用者保存寄存器 r4、r5 和 r6 的函数的序言和尾声。序言复制寄存器值至堆栈，尾声则将值从堆栈归还至寄存器。

```
# Prologue
  entsp 4
  stw r4, sp[1]
  stw r5, sp[2]
  stw r6, sp[3]

# Main body of function goes here
# ...

# Epilogue
  ldw r4, sp[1]
  ldw r5, sp[2]
  ldw r6, sp[3]
  retsp 4
```

37.4.3 资源使用

连接程序尝试计算各函数所需的资源数，包括其存储要求、线程数、通道传输结束及其使用的计时器等。这使连接程序能够检查确定最终可执行文件的资源使用不超过目的设备上可得的资源。

对函数 f，XS1 ABI 定义的资源使用符号如下所示：

图 71：XS1 ABI 定义的资源使用符号

符号	描述
f.nstackwords	堆栈大小
f.maxthreads	所分配的最大线程数，包括当前线程
f.maxchanends	所分配的最大通道传输结束数量
f.maxtimers	所分配的最大计时器数量

XMOS

你可以利用 linkset 指令定义资源使用符号。如果是全局函数，你也可以将资源使用符号做成全局的。

以下示例程序定义了函数 f 的资源使用符号，该函数 f 采用 4 字堆栈，2 条线程，0 个计时器和 2 个通道传输结束。

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 5
.globl f.maxthreads
.linkset f.maxthreads, 2
.globl f.maxtimers
.linkset f.maxtimers, 0
.globl f.maxchanends
.linkset f.maxchanends, 2
```

在更为复杂的情况下，你可以利用最大（$M）和附加（+）运算符生成资源使用表达式，由连接程序求值。如果依次调用两个方程，那么你应该计算两个方程的最大值，但如果是并行调用，你则应该计算两个方程的总和。

以下示例程序定义了函数 f 的资源使用符号，该函数 f 将堆栈扩展至 10 字，分配两个计时器，并在发布计时器和返回之前依次调用函数 g 和函数 h。

```
.globl f
.globl f.nstackwords
.linkset f.nstackwords, 10 + (g.nstackwords $M h.nstackwords)
.globl f.maxthreads
.linkset f.maxthreads, 1 + ((g.maxthreads-1) $M (h.maxthreads-1))
.globl f.maxtimers
.linkset f.maxtimers, 2 + (g.maxtimers $M h.maxtimers)
.globl f.maxchanends
.linkset f.maxchanends, 0 + (g. maxchanends $M h.maxchanends)
```

如果某资源使用符号的值未知，那么你可以省略其定义，例如，如果函数通过一个函数指针进行间接调用的情况。但是，如果需要符号的值来满足程序中的重新安置，则程序将无法链接。

37.4.4 副作用

XC 语言要求在 select 陈述中用作布尔保护信息的函数无任何副作用。此外，它还要求从某交易陈述内部调用的函数不得宣布通道传输。除非你将下列符号明确设定为零，否则默认情况下，假设函数 f 有副作用，并且会宣布通道传输。

图 72：指令副作用的符号	符号	描述
	f.locnoside	规定函数是否有副作用
	f.locnochandec	规定函数是否分配通道传输结束

37.5 删除块

连接程序可以删除未使用的编码和数据。待删除的编码和数据必须放在删除块中。在最终链接时间，如果某删除块内的所有符号均未被引用，则将该删除块从最终图像中删除。

以下示例程序宣布了一个删除块中的一个符号。

```
  .cc_top f.function, f
f:
  .cc_bottom f.function
```

.cc_top 指令和.cc_bottom 指令的首个参数是删除块的名称。.cc_top 指令具有一个附加参数，即作为块删除的基础的一个符号。如果参考了该符号，则不删除块。

各删除块都必须有一个名称，该名称在汇编文件中是唯一的。

37.6 类型字符串

一个类型字符串是指用来描述一个函数或变量的类型的字符串。类型信息编码为一个类型字符串是由 XS1 ABI 规定的。利用以下指令关联一个类型字符串和一个符号。

图 73：类型字符串指令	联编	指令
	全局	.globl name, “typestring”
	外部	.extern name, “typestring”
	局部	.locl name, “typestring”

当来自某目的文件的一个符号与另一目的中名称相同的一个符号匹配时，连接程序检查类型字符串是否兼容。如果类型字符串兼容，链接照常进行。如果类型字符串是函数类型，区别仅在于是否存在数组界参数，则连接程序将生成一个形实转换程序，并用这个形实转换程序替换符号的使用，以解释参数中的区别。出现其它任何类型字符串不匹配，连接程序均

错误。这确保了根据多个文件编制的程序与根据单个文件编制的程序一样健壮。

如果你不能为一个符号发出一个类型字符串，则假设与该符号的对比兼容。如果你正在执行一个带有未知大小数组的函数，则你应当发出一个类型字符串，以便从 C 和 XC 调用它。在其它情况下，类型字符串可以忽略，但不进行错误检查。

37.7 示例

以下程序将文字“Hello world”输出为标准输出。

```
const char str[] = "Hello world";

int main() {
  printf(str);
  return 0;
}
```

以下汇编实现符合 XS1 ABI。

```
  .extern printf, "f{si}(p(c:uc),va)"
  .section .cp.rodata, "ac", @progbits
  .globl str, "a(12:c:uc)"
  .cc_top str.data, str
  .align 4
str :
  .asciiz "Hello world"
.cc_bottom str.data
  .globl str.globound
  .set str.globound, 12

  .text
  .globl main, "f{si}(0)"
.cc_top main.function, main
  .align 2
main:
  entsp 1
  ldaw r11, cp[str]
  mov r0, r11
  bl printf
  ldc r0, 0
  retsp 0
.cc_bottom main.function
  .globl main.nstackwords
  .linkset main .nstackwords, 1 + printf.nstackwords
  .globl main.maxthreads
  .linkset main.maxthreads, printf.maxthreads
  .globl main.maxtimers
  .linkset main.maxtimers, 0 + printf.maxtimers
  .globl main.maxchanends
  .linkset main.maxchanends, 0 + printf.maxchanends
  .linkset main.locnochandec, 1
  .linkset main.locnoside, 1
```

通过定义资源使用的符号，连接程序可以检查程序是否适合某目的设备。通过提供类型字符串，连接程序可以在链接不同的目的文件时检查类型的兼容性。连接程序可以删除位于删除块中的未使用的编码和数据。

XMOS

38 带汇编语言的 XTA 使用

本章内容

采用汇编语言写入程序时，仍然需要编码，以便于使用汇编指令。

38.1 汇编指令

XMOS 计时分析仪指令将计时元数据添加至 ELF 部分。

- Xtabranch 指出了以逗号分隔的地址列表，可以自当前地址进行分支。
- Xtacall 将当前地址标记为带特定标签的函数。
- Xtaendpoint 将当前地址作为带特定标签的端点。
- Xtalabel 采用特定标签标记当前位置。
- Xtacorestart 规定了可以初始化的逻辑核心，自当前位置开始执行程序。
- Xtacorestop 规定了于当前位置执行指令的逻辑核心不得执行其他指令。

Xtacall、xtaendpoint 和 xtalabel 指令只能用于编译器，将一系列源代码与编译指令连接起来。通过使用汇编码中的标签可以使用这些指令所提供的 XTA 功能（计时除外）。

Xtacall、 xtaendpoint 和 xtalabel 中 XTA 所使用的字符串之间不得有空格。

38.2 分支表示例

如果分支表以汇编语言编写，那么必须添加分支目的信息，以便正确分析汇编语言。这些信息以 a .xtabranch 指令的形式列出，以图 74 中的编码为例。

```
. type f, @function
. globl f
f:
    entsp 1
. xtabranch Ltarget1 , Ltarget2 , Ltarget3
    bru r0
Ltarget1 :
    bl taskA
    retsp 1
Ltarget2 :
    bl taskB
    retsp 1
Ltarget3 :
    bl taskC
    retsp 1
```

图 74.设置分支目的

XMOS

XTA 无法确定 bru 指令将分支于何处，因其分离了一个寄存器值，而该寄存器为主函数的一个自变量。通过这些指令，XTA 可以认为 bru 指令有三个目的值（Ltarget1、Ltarget2、Ltarget3），XTA 可以成功对函数进行计时。

38.3 核心启停示例

默认情况下，XTA 假定初始逻辑核心可于内存中开始执行。然而，如果开发人员在汇编语言中添加其他核心，则 XTA 仍然需要添加.xtacorestart 和.xtacorestop 指令，了解哪些代码可用，以图 75 中的代码为例。

```
. type main , @function
. globl main
main :
    getr r1 , XS1 \ _RES \ _TYPE \ _CORE
    ldap r11 , secondCore
    init t[r1 ]:pc , r11
    start t[r1]
    ldc r1 , 0
loop :
    bf r1 , loop
    retsp 0

secondCore :
. xtacorestart
    ldc r0 , 1
    tsetmr r1 , r0
. xtacorestop
    freet
```

图 75. 设置核心启停点

通过 xtacorestart 和 xtacorestop 指令，XTA 知道标签“第二核心”之后的代码可用，因此可对该代码进行分析。

39 汇编语言编程手册

本章内容

XMOS 汇编语言支持对象在带有 DWARF 3[6]调试信息的可执行和与可链接格式（ELF）[7]中形成。至 ELF 格式的扩展记录在 XMOS 应用程序二进制接口中（见 ξ40）。

39.1 词法约定

一共有六种类型的令牌：符号名称、指令、常量、运算符、指令助记符及其它分隔符。忽略空格、制表符、换页符及注释，但它们是单独令牌的情况除外。

39.1.1 注释

字符#引出一个注释，该注释由换行符终止。注释不存在于字符串常量中。

39.1.2 符号名称

符号名称以一个字母或字符‘.’、‘-’或‘$’中的任意一个开头，后接一串字母、数字、句号、下划线和美元符号（任选）。大写字母和小写字母不同。

39.1.3 指令

指令以‘.’开头，后接一个或多个字母。指令指令汇编程序实施某些行为（见 ξ35.6）。

[7] http://www.xmos.com/references/dwarf3

[8] http://www.xmos.com/references/elf

39.1.4 常数

常数为一个整数，字符常数或字符串常量。

·二进制整数为 0b 或 0B，后跟零个或多个数字 01。

·八进制整数为 0，后跟零个或多个数字 01234567。

·十进制整数为非零位，后跟零个或多个数字 0123456789。

·十六进制整数为 0x 或 0X，后跟一个或多个数字和字母 0123456789abcdefABCDEF。

·字符常数是由单引号包围的字符序列。

·字符串常量是由双引号包围的字符序列。

C 转义序列可用于指定某些字符。

39.2 区段和重定位

命名 ELF 区段由命令（见§35.6.10）指定。此外，有一个唯一的未命名“绝对”区段和一个唯一的未命名“未定义”区段。记号{secname X}是指“offset X into section secname.”

绝对区段中的符号值不受重定位影响。例如，地址{absolute 0}“重定位”至运行时间地址 0。未定义区段中的符号值未设定。

汇编器保持着当前区段的轨迹。当前区段最开始设定为文本区段。命令可用于改变当前区段。可重定位存储区的汇编指令和命令在当前区段中发出。对于每个区段，汇编器维护在区段中保存当前偏移量的位置计数器。active location counter 是指当前区段的位置计数器。

39.3 符号

每个符号只有一个名称。汇编程序中的每个名称即是一个符号。本地符号是以字符“.L”开头的任何符号。本地符号可由连接器在连接不再需要时丢弃。

39.3.1 属性

每个符号都有一个 value，一个相关区段和一个 binding。符号通过 set 或 linkset 命令（见§35.6.12），或通过其在标签（见§35.4）中的应用赋值。符号在未定义区段的默认绑定为 global；所有其他符号的默认绑定则为 local。

XMOS

39.4 标签

标签是后面紧跟冒号（:）的一个符号名。符号的值设为现用位置计数器的当前值。符号的区段设为当前区段。符号名不能出现在一个以上标签中。

39.5 表达式

表达式特指一个地址或值。一个表达式的结果必须为绝对数或特定区段内的偏移量。如果表达式的所有符号均被定义且其求值结果为一个常数，则该表达式为 constant expression。如果表达式为一个常数表达式，一个符号或一个符号+-一个常数之一，则该表达式为简单表达式。表达式可在 ELF-扩展表达式区段中进行编码且由连接器计算其值（见§35.6.12）；编码方案由 ABI 确定。表达式的语法为：

```
expression   ::= unary-exp
               | expression infix-op unary-exp
               | unary-exp ? unary-exp $: unary-exp

unary-exp    ::= argument
               | prefix-op unary-exp

argument     ::= symbol
               | constant
               | ( expression )

infix-op     ::= one of
                 + - < > <= >= || << >> * $M $A & /

prefix-op    ::= one of
                 - ~ $D
```

当 section 是命名区段，绝对区段或未定义区段之一时，符号的求值结果为{section x }，且 x 是有符号的二进制补码 32 位整数。

插入运算符有与 C 相同的优先级和行为，且有相同优先级的运算符从左至右执行，此外，$M 运算符有最低优先级，而$A 运算符有最高优先级。

对于+和-运算符，图 56 中提供有效运算和结果的集。对于$D 运算符，参数必须为符号；结果为 1，当符号被定义时；否则为 0。

运算符 左操作数 右操作数 结果

Op	Left Operand	Right Operand	Result
+	{*section* x}	{absolute y}	{*section* x+y}
+	{absolute x}	{*section* y}	{*section* x+y}
+	{absolute x}	{absolute y}	{absolute x+y}
-	{*section* x}	{*section* y}	{absolute x-y}
-	{*section* x}	{absolute y}	{*section* x-y}
-	{absolute x}	{absolute y}	{absolute x-y}

图 76： + 和 -运算符的有效运算

?运算符用于从符号中选择：如果第一操作数为非零，则结果为第二操作数，否则结果为第三操作数。

对于所有其他运算符，两个参数必须为绝对数且结果也为绝对数。$M 运算符返回两个操作数的最大值，$A 运算符则返回与第二操作数对齐的第一操作数的值。

当需要绝对表达式时，如果省略，则假定{absolute 0}。

运算符	结果
$overlay_region_ptr	含重叠的重叠区虚拟地址
$overlay_index	重叠区的重叠指数
$overlay_physical_addr	重叠的物理地址
$overlay_virtual_addr	重叠的虚拟（运行）地址
$overlay_num_bytes	重叠尺寸（以字节计）

图 77. 重叠性质询问函数

对于其他运算符，两个自变量必须为绝对数，而结果也必须为绝对数。$M 运算符返回两个操作数中的较大值， $A 运算符返回与第二个操作数对齐的第一个操作数的值。

当需要绝对表达式时，如果省略，则假定{absolute 0}。

39.6 命令

命令指令汇编器执行某一操作。本节列出了支持的命令。

39.6.1 add_to_set

add_to_set 指令将表达式加入一套与关键符号相关的表达式中，其句法结构如下：

```
add-to-set-directive    ::= .add_to_set symbol , expression
                          | .add_to_set symbol , expression , symbol
```

可选谓语符号可作为第 3 个自变量。如果给出了该自变量，则只有当谓语符号未从链接对象中除去时，才会向这套表达式中添加表达式。

39.6.2 max_reduce, sum_reduce

max_reduce 指令计算一套表达式中的最大值；sum_reduce 则计算一套表达式值的和。

39.6.3 align

align 命令将现用位置计数器填补至指定存储边界，其语法为：

align-directive ::= .align expression

表达式必须为常数表达式，其值必须为 2 的幂。该值指定以字节的单位的所需对齐。

39.6.4 ascii, asciiz

ascii 命令将每个字符串汇编至连续地址中。asciiz 命令也一样，除非每个字符串后跟一个空字节。

```
ascii-directive    ::= .ascii string-list
                     | .asciiz string-list

string-list        ::= string-list , string
                     | .asciiz string
```

39.6.5 byte, short, int, long, word

这些命令为每个表达式发出一个数字，即在运行时间该表达式的值。字节顺序由目的架构的字节顺序决定。用字类命令发出的数字的大小由目的架构上自然字类的大小决定。用其他命令发出的数字的大小由 ABI 中相应类型的大小决定。

value-directive	::=	*value-size exp-list*
value-size	::=	.byte
	|	.short
	|	.int
	|	.long
	|	.word
exp-list	::=	*exp-list* , *expression*
	|	*expression*

39.6.6 file

file 命令有两种形式。

file-directive	::=	.file *string*
	|	.file *constant string*

当与一个参数一起使用时，file 命令通过类型 STT_FILE 和指定字符串值创建一个 ELF 符号表条目。该条目确保为字符表中的第一条目。

当与两个参数一起使用时，file 命令添加一个条目至 DWARF 3.debug_line 文件名表格中。第一个参数为唯一的正整数，用作表中条目的索引。第二个参数为文件名。

39.6.7 loc

.loc 命令添加一行至 DWARF 3 .debug_line 行号矩阵中。

loc-directive	::=	*constant constant constant*$_{opt}$
	|	*constant constant constant* (*loc-option*)*
loc-option	::=	basic_block
	|	prologue_end
	|	epilogue_begin
	|	is_stmt *constant*
	|	isa *constant*

XMOS

地址寄存器设为现用位置计数器。开始两个参数分别设定文件和行寄存器。可选的第三个参数设定列寄存器。其他参数设定.debug_line 状态机中的更多寄存器。

basic_block

将 basic_block 设置为真

prologue_end

将 prologue_end 设置为真

epilogue_begin

将 epilogue_begin 设置为真

is_stmt

将 is_stmt 设置为规定值，必须是 0 或 1。

isa

将 isa 设置为规定值

39.6.8 weak

weak 命令设定指定符号上的弱属性。

```
weak-directive   ::=  .weak symbol
```

39.6.9 globl, global, extern, locl, local

globl 命令使指定符号在连接期间可被其他对象看见。extern 命令特指符号在另一个对象中定义。locl 命令特指符号具有本地绑定，

```
visibility     ::=  .globl
                 |  .extern
                 |  .locl
                 |  .global
                 |  .extern
                 |  .local

vis-directive  ::=  visibility symbol
                 |  visibility symbol , string
```

如果提供可选字符串，在包含符号和字符串表格索引的 ELF 扩展型区段中创建 SHT_TYPEINFO 条目，所述字符串表格的条目包含指定字符串。(如果字符串已不存在于字

符串表格中，则其被插入。)该字符串的意义由 ABI 决定。global 和 local 命令为 globl 和 locl 命令的同义词，其用于兼容其他汇编器。

XMOS

39.6.10 typestring

typestring 添加一个 SHT_TYPEINFO 条目至包含符号和字符表格索引的 ELF 扩展型区段中，所述字符表格索引的条目包含指定字符串。（如果字符串已不存在于字符串表格中，则其被插入。）该字符串的意义由 ABI 决定。

typestring-directive ::= .typestring symbol , string

39.6.11 ident, core, corerev

这些命令每个都创建一个 ELF 备注区段，命名为“.xmos_note.”

info-directive ::= .ident string

| .core string

| .corerev string

该区段的内容为一个（名称，类型，值）三元组：名称为 xmos；类型为 IDENT, CORE 或 COREREV；值为指定字符串。

39.6.12 section, pushsection, popsection

区段命令改变当前 ELF 区段（见§35.2）。

section-directive ::= sec-or-push name

| sec-or-push name , flags sec-typeopt

| .popsection

sec-or-push ::= .section

| .pushsection

flags ::= string

sec-type ::= type

| type , flag-args

type ::= @progbits

| @nobits

flag-args ::= string

XMOS

紧跟 section 或 pushsection 命令的编码汇编且增补至命名区段。可选标记可包含下列字符的任意组合。

a 区段为可分配的

c 区段位于全局常数存储库中

d 区段位于全局数据区中

w 区段为可写入的

x 区段为可执行的

M 区段为可合并的

S 区段包含以零结尾的字符串

可选类型参数 progbits 特指包含数据的区段；nobits 特指不包含的区段。

如果 M 符号指定为标记，类型参数必须指定且必须提供一个整数作为标记特定参数。标记特定参数代表区段中数据条目的实体尺寸。例如：

.section .cp.const4, "M", @progbits, 4

有 M 标记但无 S 标记的区段必须包含固定大小的常数，每个为 flag-args 字节长。有 M 和 S 标记的区段必须包含以零结尾的字符串，每个字符为 flag-args 字节长。连接器可除去区段内有相同名称，实体大小和标记的重复条目。

名称相同的每个区段必须有相同类型和标记。section 命令用指定区段替代当前区段。pushsection 命令推动当前区段至 section stack 的顶部，然后用指定区段取代当前区段。popsection 命令用区段堆叠顶部的区段替代当前区段，然后从堆叠中取出该区段。

39.6.13 text

text 命令将当前 ELF 区段变为.text 区段。区段类型和属性由 ABI 决定。

text-directive ::= .text

39.6.14 set, linkset

符号通过 set 或 linkset 命令赋值。

set-directive ::= set-type symbol , expression

set-type ::= .set

| .linkset

XMOS

set 命令用表达式的值定义命名符号。表达式必须为一个常数或一个符号；如果表达式是一个常数，符号定义在绝对区段中；如果表达式是一个符号，定义符号继承该符号其区段信息和其他属性。

linkset 命令也一样，除非表达式未确定值；反而，在 ELF 扩展表达式区段中创建一个或多个 SHT_EXPR 条目，共同形成一个表达式树表示。

汇编码中所用的任何符号可能是 SHT_EXPR 条目的目的，在这种情况下，其值由连接器在表达式中所有其他符号的值已知时通过确定表达式的值来计算。这可在任何增量连接阶段发生。一旦值已知，则可设定符号的值且将表达式条目从连接对象中除去。

39.6.15 cc_top, cc_bottom

cc_top 和 cc_bottom 命令用于标记消除块的开始和结尾。

cc-top-directive	::=	.cc_top *name* , *predicate*
	|	.cc_top *name*
cc-directive	::=	*cc-top-directive*
	|	.cc_bottom *name*
name	::=	*symbol*
predicate	::=	*symbol*

名称相同的 cc_top 和 cc_bottom 命令是指相同的消除块。一个消除块必须只有一个 cc_top 命令和一个 cc_bottom 命令。消除块的顶部和尾部必须在相同区段内。消除块包含该区段内 cc_top 和 cc_bottom 命令见的数据和标签。消除块不可相交。消除块重叠是非法的。

如果下列之一为真，消除块最终仍然是可执行的：

·消除块内的标签引用自消除块外的位置。

·消除块内的标签引用自未消除的消除块。

·谓词符号定义在消除块外或包含在未消除消除块内。

如果这些条件均不为真，则消除块从最终可执行文件中删除。

XMOS

39.6.16 scheduling

scheduling 命令启用或禁用指令排程。当启用排程时，汇编器可重新排列指令以最小化 FNOP 的数目。默认排程模式由命令行选项-fschedule (见 §9.4)决定。

```
scheduling-directive   ::= .scheduling

scheduling-mode        ::= on
                         | off
                         | default
```

39.6.17 syntax

syntax 命令改变当前语法模式。在每个模式中如何指定汇编指令详见§35.7。

```
syntax-directive   ::= .syntax syntax

syntax             ::= default
                     | architectural
```

39.6.18 assert

assert-directive ::= .assert constant , symbol , string

assert 命令在产生可执行对象前需要一个待测试断言：如果符号有非零值，断言失败。如果常数为 0，失败应报告为警告；如果常数为 1，失败应报告为错误。字符串是汇编器或连接器发出的失败消息。

39.6.19 重叠指令

重叠指令控制运行时根据需要加载的重叠处代码和数据如何分隔。

```
overlay-directive   ::= .overlay_reference symbol , symbol
                      | .overlay_root symbol , symbol
                      | .overlay_root symbol
                      | .overlay_subgraph_conflict sym-list

sym-list            ::= sym-list , symbol
                      | symbol
```

重叠根指令确定了需要作为重叠根的首个符号。可选的第二个符号则规定了重叠关键符号。如果未明确规定重叠关键符号，则应采用重叠根符号作为关键符号。为多个重叠根制定同一个重叠关键符号，则可以使重叠根位于同一个重叠中。

overlay_reference 指令指出，链接器需假定链接器将项目划分为不同的重叠时，第一个符号与第二个符号之间能互相参照。

overlay_subgraph_conflict 指令指出，链接器不得将可用编码或数据放置在映射到重叠

XMOS

区（该重叠区包含来自其他符号的可用编码或数据）的重叠上。

39.6.20 语言命令

语言命令在 ELF 扩展表达式区段中创建条目；编码由 ABI 确定。

```
xc-directive  ::= globdir symbol , string
               |  globdir symbol , symbol , range-args , string
               |  .globpassesref symbol , symbol , string
               |  .call symbol , symbol
               |  .par symbol , symbol , string

range-args    ::= expression , expression


globdir       ::= .globread
               |  .globwrite
               |  .parwrite
               |  .globpassesref
```

对于每个命令，字符串是汇编器或连接器在遇到由命令引起的错误时显示的消息。

call

两个符号必须有函数类型。该命令设置属性，即第一函数可调用第二函数。

par

两个符号必须有函数类型。该命令设置属性，即第一函数与第二函数并行调用。

globread

第一符号必须有函数类型且第二命令必须有对象类型。该命令设置属性，即函数可读取对象。当指定一范围，第一表达式是读取地址的变量字节的开始偏移量，而第二表达式则是读取字节的大小。

globwrite

第一符号必须有函数类型且第二命令必须有对象类型。该命令设置属性，即函数可写入对象。当指定一范围，第一表达式是写入地址的变量字节的开始偏移量，而第二表达式则是写入字节的大小。

parwrite

第一符号必须有函数类型且第二命令必须有对象类型。该命令设置属性，即在写入和功能调用的求值顺序未定义时写入对象的表达式中调用函数。当指定一范围，第一表达式是写入地址的变量字节的开始偏移量，而第二表达式则是写入字节的大小。

globpassesref

第一符号必须有函数类型且第二命令必须有对象类型。该命令设置属性，即可引用函数来传递对象。

39.6.21 XMOS 计时分析器命令

XMOS 计时分析器命令添加计时数据库至 ELF 区段中。

```
xta-directive    ::= .xtabranch exp-list_opt
                   | .xtaendpoint string , source-location
                   | .xtacall string , source-location
                   | .xtalabel string , source-location
                   | .xtathreadstart
                   | .xtathreadstop
                   | .xtaloop constant
                   | .xtacommand string , source-location

source-location  ::= string , string , constant
```

源位置的第一字符串为编译目录。第二字符串为文件路径。路径可指定为自编译目录起的相对路径或绝对路径。第三参数为行号。

·xtabranch 特指可从当前位置分支或分支至当前位置的位置逗号分隔列表。

·xtaendpoint 用指定标签将当前位置标记为终点。

·xtacall 用指定标签将当前位置标记为函数调用。

·xtathreadstart 特指可初始化以在当前位置开始执行的线程。

·xtathreadstop 特指在当前位置执行指令的线程将不执行任何进一步指令。

·xtaloop 特指包含当前位置的最内层循环执行指定的次数。

·xtacommand 特指在分析可执行性时待执行的 XTA 命令。

39.6.22 uleb128, sleb128

下列命令为表达式的逗号分隔列表中的每个表达式发出一个值，编码有符号的或无符号的 DWARF 小端基 128 数。

leb-directive ::= .uleb128 exp-list

| .sleb128 exp-list

XMOS

39.6.23 space, skip

Space 命令发出字节序列，由第一表达式指定，有填充值的每个字节序列由第二表达式指定。两个表达式必须为常数表达式。

```
space-or-skip     ::= .space
                    | .skip

space-directive   ::= space-or-skip expression
                    | space-or-skip expression , expression
```

skip 命令是空格命令的同义词，其用于兼容其他汇编器。

39.6.24 类型

type 命令特指符号类型，为函数类型或对象类型。

```
type-directive   ::= .type symbol , symbol-type

symbol-type      ::= @function
                   | @object
```

39.6.25 尺寸

尺寸命令特指与符号相关的尺寸。

size-directive ::= .size symbol , expression

39.6.26 jmptable, jmptable32

Jmptable 和 jmptable32 命令生成一个无条件分支指令表。每个分支指令的目的为列表中的下一个标签。每个分支指令的尺寸为 jmptable 命令 16 位，jmptable32 命令 32 位。

jmptable-directive ::= .jmptable jmptable-listopt

| .jmptable32 jmptable-listopt

jmptable-list ::= symbol

| jmptable-list symbol

每个符号必须为一个标签。最多可指定 32 个标签。如果无条件分支距离不适于 16 位分支指令，分支可作为表格末尾的弹簧，执行到目的标签的分支。

39.7 指令

汇编指令通常被插入 ELF 代码段。指令语法为：

instruction	::=	*mnemonic instruction-args*$_{opt}$
instruction-args	::=	*instruction-args* , *instruction-arg*
	|	*instruction-arg*
instruction-arg	::=	*symbol* [*expression*]
	|	*symbol* [*expression*] : *symbol*
	|	*expression*

汇编指令总结如下，引用 XS1 架构手册（见 X7879)。本册中，构架格式为记录型。语法指令用来切换到此编码模式。

使用以下符号：

bitp 1, 2, 3, 4, 5, 6, 7, 8, 16, 24 及 32 其中之一

b 以寄存器为基址

c 以寄存器为条件运算元

d, e 以寄存器为目的运算元

i 以寄存器为索引运算元

r 以寄存器为资源标示

s 以寄存器为源运算元

t 以寄存器为线程标示符

us 小的无符号常量从 0 至 11

ux 无符号常量 0...(2x -1)

v, w, x, y 寄存器用于两个及以上源运算元

寄存器为 r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, sp, dp, cp 和 lr 其中之一，该指令决定了哪一个指令被允许。

只要有指令格式可选择，汇编器选择尺寸最小的格式。以强制一个特定格式、指定 INSTRUCTION_format 形式的助记符与架构手册中描述的指令与格式名称一致。例如，LDWCP_ru6 助记符指定 LDWCP 指令的 ru6 格式。

下表为 XS1 架构手册中指令的页面数（见 X7879)。

XMOS

335.7.1	数据存取			
助记符	运算元	规格	语义	页数
ld16s	d, b[i]	16	加载符号 16 位	121
ld8u	d, b[i]	16	加载无符号 8 位	122
lda16	d, b[i]	32	增加 16-bit 地址	123
lda16	d, b[-i]	32	减去 16-bit 地址	124
ldap	r11, u20	16/32	加载 pc-relative 地址	125
ldap	r11, -u20	16/32	加载 pc-relative 地址	126
ldaw	d, b[i]	32	增加一个字地址	131
ldaw	d, b[-i]	32	删减一个字地址	127
ldaw	d, b[us]	32	即刻增加一个字地址	132
ldaw	d, b[-us]	32	即刻删减一个字地址	128
ldaw	r11, cp[u16]	16/32	在常量池加载字地址	129
ldaw	d, dp[u16]	16/32	在数据池加载字地址	130
ldaw	d, sp[u16]	16/32	在堆栈加载字地址	133
ldw	et, sp[4]	16	从堆栈加载 ET	135
ldw	sed, sp[3]	16	从堆栈加载 SED	137
ldw	spc, sp[1]	16	从堆栈加载 SPC	138
ldw	ssr, sp[2]	16	从堆栈加载 SSR	139
ldw	d, b[i]	16	加载字	140
ldw	d, b[us]	16	即刻加载字	141
ldw	d, cp[u16]	16/32	从常量池加载字	142
ldw	r11, cp[u20]	16/32	从常量池加载字	143
ldw	d, dp[u16]	16/32	从数据池加载字	144
ldw	d, sp[u16]	16/32	从堆栈加载字	145
set	cp, s	16	设置常量池	175
set	dp, s	16	设置数据指令器	177
set	sp, s	16	设置堆栈指令器	185
st16	s, b[i]	32	16-bit 存储	196
st8	s, b[i]	32	8-bit 存储	197
stw	sed, sp[3]	16	在堆栈存储 SED	199
stw	et, sp[4]	16	在堆栈存储 ET	198
stw	spc, sp[1]	16	在堆栈存储 SPC	200
stw	ssr, sp[2]	16	在堆栈存储 SSR	201
stw	s, b[i]	32	存储字	202
stw	s, b[us]	16	即刻存储字	203
stw	s, dp[u16]	16/32	在数据池存储字	204
stw	s, sp[u16]	16/32	在堆栈存储字	205

39.7.2 支化、跳跃及调用

助记符	运算元	规格	语义	页数
bau	s	16	绝对无条件支化	53
bf	c, u16	16/32	如为否，相对支化	63
bf	c, -u16	16/32	如为否，相对支化	60
bl	u20	16/32	相对支化与连接	59
bl	0	16/32	相对支化与连接	58
bla	s	16	通过寄存器绝对支化与连接	55
bla	cp[u20]	16/32	通过 CP 绝对支化与连接	56
blat	u16	16/32	通过表格绝对支化与连接	57
bru	s	16	通过寄存器无条件相对支化	66
bt	c, u16	16/32	如为是，相对支化	64
bt	c, -u16	16/32	如为是，相对支化	61
bu	u16	16/32	无条件相对支化	65
bu	0	16/32	无条件相对支化	62
entsp	u16	16/32	调整堆栈并储存连接寄存器	90
extdp	u16	16/32	扩展数据指令器	93
extsp	u16	16/32	扩展堆栈指令器	94
retsp	u16	16/32	返还	169

39.7.3 数据处理

助记符	运算元	规格	语义	页数
add	d, x, y	16	增加	47
add	d, x, us	16	即刻增加	48
and	d, x, y	16	位元 and	49
andnot	d, s	16	与非	50
ashr	d, x, y	32	算术右移	51
ashr	d, x, bitp	32	即刻算术右移	52
bitrev	d, s	32	位倒序	54
byterev	d, s	32	字节倒序	67
clz	d, s	32	增加了前导零计数	73
crc32	d, r, p	32	字 CRC	74
crc8	r, o, d, p	32	8-step CRC	75
divs	d, x, y	32	符号除法	79
divu	d, x, y	32	无符号除法	80
eq	c, x, y	16	均等	91
eq	c, x, us	16	即刻均等	92
ladd	e, d, x, y, v	32	长的无符号进位添加	120

助记符	运算元	规格	语义	页数
ldc	d, u16	16/32		134
ldivu	d, e, v, x,y	32	长的无符号划分	136
lmul	d, e, x, y,v, w	32	长的叠加	146
lss	c, x, y	16	少于符号	147
lsu	c, x, y	16	少于无符号	148
lsub	e, d, x, y,v	32	长的无符号差集	149
maccs	d, e, x, y	32	叠加积累符号	150
maccu	d, e, x, y	32	叠加积累无符号	151
mkmsk	d, s	16	制作掩码	153
mkmsk	d, bitp	16	立即制作掩码	154
mul	d, x, y	32	叠加	156
neg	d, s	16	二进制补码无效	157
not	d, s	16	按位否	158
or	d, x, y	16	按位或	159
rems	d, x, y	32	符号剩余	167

39.7.4 并发及线程同步

助记符	运算元	规格	语义	页数
freet	16	免费同步线程	96	
get	r11, id	16	获取线程 ID	100
getst	d, res[r]	16	获取同步线程	108
mjoin	res[r]	16	主同步并加入	152
msync	res[r]	16	主同步	155
ssync	16	从同步	195	
init	t[r]:cp, s	16	初始化线程的 CP	212
init	t[r]:dp, s	16	初始化线程的 DP	213
init	t[r]:lr, s	32	初始化线程的 LR	214
助记符	运算元	规格	语义	页数
init	t[r]:pc, s	16	初始化线程的 PC	215
init	t[r]:sp, s	16	初始化线程的 SP	216

set	t[r]:d, s	16	在线程中设置储存器	218
start	t[r]	16	启动线程	219
tsetmr	d, s	16	主线程中设置储存器	217

39.7.5 通信

助记符	运算元	规格	语义	页数
chkct	res[r], s	16	控制符记测试	68
chkct	res[r], us	16	立即控制符记测试	69
getn	d, res[r]	32	获取网络	103
in	d, res[r]	16	输入数据	110
inct	d, res[r]	16	输入控制符记	111
int	d, res[r]	16	输入数据标记	114
out	res[r], s	16	输出数据	160
outct	res[r], s	16	输出控制符记	161
outct	res[r], us	16	立即输出控制标记	162
outt	res[r], s	16	输出数据符记	165
setn	res[r], s	32	设置网络	180
testlcl	d, res[r]	32	测试本地	209
testct	d, res[r]	16	控制符记测试	210
testwct	d, res[r]	16	控制符记位置测试	211

39.7.6 资源操作

助记符	运算元	规格	语义	页数
clrpt	res[r]	16	明确端口时间	71
endin	d, res[r]	16	结束当前输入	89
freer	res[r]	16	发布资源	95
getd	d, res[r]	32	获取资源数据	97
getr	d, us	16	分配资源	105
getts	d, res[r]	16	获取端口时间标记	109
in	d, res[r]	16	输入数据	110
inpw	d, res[r], bitp	32	输入部分字	112
inshr	d, res[r]	16	输入及右移	113
out	res[r], s	16	输出数据	160
outpw	res[r], s, bitp	32	输出部分字	163

XMOS

outshr	res[r], s	16	输出数据并移位	164
助记符	运算元	规格	语义	页数
peek	d, res[r]	16	查看端口数据	166
setc	res[r], s	32	设置资源控制位	172
setc	res[r], u16	16/32	立即设置资源控制位	170
setclk	res[r], s	32	为资源置位时钟	174
setd	res[r], s	16	设置数据	176
setev	res[r], r11	16	设置环境矢量	178
setpsc	res[r], s	16	设置端口移数	182
setpt	res[r], s	16	设置端口时间	183
setrdy	res[r], s	32	设置端口准备输入	184
settw	res[r], s	32	设置端口传输端口	187
setv	res[r], r11	16	设置项目向量	188
syncr	res[r]	16	同步资源	208

39.7.7 事件处理

助记符	运算元	规格	语义	页数
clre	16	清除所有项目	70	
clrsr	u16	16/32	清除 SR 中的位	72
edu	res[r]	16	禁用项目	85
eef	d, res[r]	16	如为否，启用项目	86
eet	d, res[r]	16	如为是，启用项目	87
eeu	res[r]	16	启用项目	88
getsr	r11, u16	16/32	从 SR 中获取位	107
setsr	u16	16/32	在 SR 中设置位	186
waitef	c	16	如为否，延期执行项目	220
waitet	c	16	若为是，延期执行项目	221
waiteu	16	延期执行项目	222	

39.7.8 中断、异常及内核调用

助记符	运算元	规格	语义	页数
clrsr	u16	16/32	清除 SR 中的位	72
ecallf	c	16	如为否，引发异常	83

XMOS

ecallt	c	16	如为是，引发异常	84
get	r11， ed	16	将 ED 纳入 r11	98
get	r11， et	16	将 ET 纳入 r11	99
get	r11， kep	16	获取内核入口点	101
get	r11， ksp	16	获取内核堆栈储存器	102
getsr	r11， u16	16/32	从 SR 获取 bits	107

助记符	运算元	规格	语义	页数
kcall	s	16	内核调用	115
kcall	u16	16/32	立即内核调用	116
kentsp	u16	16/32	切换至内核堆栈	117
krestsp	u16	16/32	从内核堆栈恢复堆栈指针	118
kret	16	内核返还	119	
set	kep， r11	16	设置内核入口点	179
setsr	u16	16/32	在 SR 中设置 bits	186

39.7.9 调试

助记符	运算元	规格	语义	页数
dcall	16	导致调试中断	76	
dentsp	16	保存并修改堆栈储存器进行调试	77	
dgetreg	s	16	调试另一个线程储存器的读取	78
drestsp	16	恢复非调试堆栈指令器	81	
dret	16	从调试中断返还	82	
get	d， ps[r]	32	获取处理器状态	104
set	ps[r]， s	32	设置处理器状态	181

39.7.10 伪指令

默认语法模式中，汇编器支持一小组伪指令。此类指令不存在于处理器上，但会经汇编器翻译进入 XS1 指令。

mov	运算元	定义
dcall	d， s	加入 d, S, 0
nop		r0, r0, 0

39.8 汇编程序

汇编程序由语句序列组成。

program	::=	⟨*statement*⟩*
statement	::=	*label-list*$_{opt}$ *dir-or-inst*$_{opt}$ *separator*
label-list	::=	*label*
	|	*label-list label*
dir-or-inst	::=	*directive*
	|	*instruction*
separator	::=	*newline*
	|	;
directive	::=	*align-directive*
	|	*ascii-directive*
	|	*value-directive*
	|	*file-directive*
	|	*loc-directive*
	|	*weak-directive*
	|	*vis-directive*
	|	*text-directive*
	|	*set-directive*
	|	*cc-directive*
	|	*scheduling-directive*
	|	*syntax-directive*
	|	*assert-directive*
	|	*xc-directive*
	|	*xta-directive*
	|	*space-directive*
	|	*type-directive*
	|	*size-directive*
	|	*jmptable-directive*

第 N 部分 XS1 设备编程

内容

40 XS1 设备的 XCC 目的依赖型行为

本章内容

-时钟模块支持

-端口支持

-信道通讯

本章描述了适用于 XS1 架构的 XMOS 编译器套装行为。

40.1 时钟模块支持

XS1 设备提供单个基准时钟，以外部振荡器产生的频率运转。XC 要求系统设计员保证基准时钟以 100MHz 的频率运转，以确保计时器正常工作。

xCORE 模块提供了一整套可编程时钟模块，可用来产生时钟信号。

名为< xs1.h >的头文件对时钟类型进行了界定。extern 中没有相关声明时，必须采用时钟模块表达式对时钟变量进行初始化，例如：

```
clock c = XS1_CLKBLK_1;
```

设备数据表中列出了可用的时钟模块数。其名称如上述声明，从 1 开始按顺序对其编号。

XC 中时钟为高精度时钟，其附加规则如下：

· 结构中描述了这类时钟的构成部分，只能通过外部声明来描述带该类型时钟的结构变量。

· 变量声明加前缀“on”可以表示该类型时钟对象，该类型时钟不能描述自动变量。

40.2 端口支持

XC 端口说明

XC 端口说明描述了端口 p 的源端口。在 XS1 设备中，所有用于数据输入输出的端口都通过一个 100MHz 的基准时钟（见§40.1）计时，并使用单入口缓冲器，即使端口说明并不符合缓冲关键词。

图 78 中表格根据 XC 说明是否符合缓冲关键词，来确定 XS1 端口上支持哪些输入/输出操作。

图 78：XS1 端口上支持的输入、输出操作	模式	操作		
		序列化	选通	@ 时间
	不符合	×	×	×
	缓冲	√	√	√

编译器监测并发布下列各项中出现的故障：

· 序列化：使用与端口宽度不同的输送宽度来描述不符合缓冲的端口。

· 选通：配置不符合缓冲的端口，使用即入即出信号。

· 同时使用@和 when 的输入语句：输入语句不符合缓冲时，将同时使用这两种运算符

40.2.1 序列化

```
par {
  p @ t <: x;
  q @ t :> y;
}
```

注意使用序列化时，计时输入语句规定的时间记录了最后一比特数据取样时间。由于该结构使 q 上的输入完成时立即开始 p 上的输出，因此使用序列化时会产生出人意料的结果。

为并行输入、输出该数据，将传输宽度除以端口宽度得到的数值在软件中补偿输入时间。

40.2.2 时间戳

输入语句记录的时间戳一般出现在数据取样之后，这是因为 XS1 为数据输入和时间戳输入分别提供不同的指令，所以下一数据取样之后才能输入时间戳。这一问题同样影响输出，但不会影响在选择语句保护下执行的输入。输入或输出指令执行之后，编译器会立即输入时间戳，因此在实际中，本行为很少看到。

40.2.3 缓冲端口方向改变

试图改变符合缓冲的端口的方向会产生以下行为。

40.3 信道通讯

在一些版本的 XS1 结构中，从选择语句防护下的数据流信道中输入 32 位以下的数据是不可能的。

· 编译 XS1-G 结构时，编译器不允许在 XC 数据流信道中选择长度短于 1 个字长的输入语句。命令行选项-fsubword-select 放松了这一限制，但将导致即使信道上数据可用，却不会发生具有这类函数的案例。

· 编译 XS1-G 结构时，inuchar_byref、inct_byref 和 testct 函数不得用于 XC 选择语句。命令行选项-fsubword-select 放松了这一限制，但将导致即使信道上数据可用，却不会发生具有这类函数的案例。

41 XS1 数据类型

语言中没有规定 C 和 XC 数据类型的尺寸和对正，这可以将整数尺寸设定为目的设备的自然字长，确保计算中的高性能。这同样可以对数据调整设定，使其足够宽以保证有效的内存负荷和存储。图 79 描绘了 xCORE 应用二进制界面（见§44）所规定的数据类型尺寸和对正，为 C 和 XC 编译对象的链接提供了标准界面。

	数据类型	尺寸（比特）	调整（比特）	支持		含义
				XC	C	
图 58 XS1 设备上的数据类型尺寸与对正	Char	8	8	√	√	字符类型
	Short	16	16	√	√	短整数
	Int	32	32	√	√	初始整数
	Long	32	32	√	√	长整数
	Long long	64	32	×	√	长长整数
	Float	32	32	×	√	32 位 IEEE 浮点
	Double	64	32	×	√	64 位 IEEE 浮点
	Long double	64	32	×	√	数据指令器
	Void *	32	32	×	√	端口
	Port	32	32	√	×	计时器
	Timer	32	32	√	×	信道端
	chanend	32	32	√	×	

另外：

· Char 类型默认无符号。

· 可于位字段语句中规定 char、short 和 int 数据类型。

· 零宽度位字段会一直填充，直到下一位偏移与位字段说明类型调整。

· 端口概念传输类型为无符号整数（32 位）。

· 端口概念计数器类型为无符号短整数（16 位）。
· 计时器概念计数器类型为无符号整数（32 位）。

XMOS

42 XS1 端脚映射

XS1 设备中，通过端口将引脚与外部组件相连接，从而组成通向其他设备的链路，并于链路上建立信道。端口分多路传输，可配置引脚用于不同宽度的端口。图 59 中显示了 XS1 端脚的映射，解释如下：

· 各引脚以 XnDpq 的格式命名，其中 n 为设备的有效 xCORE 数据块，而 pq 存在于表格中。引脚的物理位置取决于程序包，列在设备数据表中。

· 每条链路以字母 A-D 标识；链路线通过上标数字 0-4 来标识。

· 每个端口使用其宽度（第一个数值 1、4、8、16 或 32）和区别同一宽度多个端口的字母（A-P）来标识。这些名称对应于头文件<xs1.h>中的端口指令器（例如端口 1A 对应于指令器 XS1_PORT_1A）。端口各数据位使用上标数字 0-31 来标识。

·表格分为 6 列（或行）。前四行提供了 1 位、4 位和 8 位数据端口，而后两行则启用单一 31 位端口。不同封装选项可以输出不同行的数据；16 位和 32 位端口不适用于小设备。

程序使用的端口由一组 XC 端口说明来确定。例如，端口 p = XS1_PORT_1A 中数据块[0]：于 xCORE 数据块 0 上使用 1 位端口 1A，其中，xCORE 数据块 0 与 X0D00 引脚相连。

通常设计人员应确保说明端口引脚之间没有重叠，但必须根据需要设定优先级，当较窄端口重叠时，应使用较宽的端口。表格左侧的端口优先于右侧的端口。如果两个说明的端口共用一个引脚，则较窄的端口为先，例如：

```
on tile[2] : out port p1 = XS1_PORT_32A;
on tile[2] : out port p2 = XS1_PORT_8B;
on tile[2] : out port p3 = XS1_PORT_4C;
```

在本案例中：

·端口 p1 上的输入/输出使用 X2D02 至 X2D09 和 X2D49 至 X2D70。

·端口 p2 上的输入/输出使用引脚 X2D16 至 X2D19；自 p2 输入会于数位 0、1、6、7 上产生未定义值。

·端口 p3 上的输入/输出使用 X2D14、X2D15、X2D20 和 X2D21；自 p1 输入会于数位 28-31 上产生未定义值，此时这些数位数据输出时无驱动。

图 80：各引脚上可用的端口和链路

最高← 最低→						优先级
引脚	链路	1 位端口	4 位端口	8 位端口	16 位端口	32 位端口
XnD00		1A				
XnD01	A^4 in/out	1B				
XnD02	A^3 in/out		$4A^0$	$8A^0$	$16A^0$	$32A^{20}$
XnD03	A^2 in/out		$4A^1$	$8A^1$	$16A^1$	$32A^{21}$
XnD04	A^1 in/out		$4B^0$	$8A^2$	$16A^2$	$32A^{22}$
XnD05	A^0 in/out		$4B^1$	$8A^3$	$16A^3$	$32A^{23}$
XnD06	A^0 out/in		$4B^2$	$8A^4$	$16A^4$	$32A^{24}$
XnD07	A^1 out/in		$4B^3$	$8A^5$	$16A^5$	$32A^{25}$
XnD08	A^2 out/in		$4A^2$	$8A^6$	$16A^6$	$32A^{26}$
XnD09	A^3 out/in		$4A^3$	$8A^7$	$16A^7$	$32A^{27}$
XnD10	A^4 out/in	1C				
XnD11		1D				
XnD12		1E				
XnD13	B^4 in/out	1F				
XnD14	B^3 in/out		$4C^0$	$8B^0$	$16A^8$	$32A^{28}$
XnD15	B^2 in/out		$4C^1$	$8B^1$	$16A^9$	$32A^{29}$
XnD16	B^1 in/out		$4D^0$	$8B^2$	$16A^{10}$	
XnD17	B^0 in/out		$4D^1$	$8B^3$	$16A^{11}$	
XnD18	B^0 out/in		$4D^2$	$8B^4$	$16A^{12}$	
XnD19	B^1 out/in		$4D^3$	$8B^5$	$16A^{13}$	
XnD20	B^2 out/in		$4C^2$	$8B^6$	$16A^{14}$	$32A^{30}$
XnD21	B^3 out/in		$4C^3$	$8B^7$	$16A^{15}$	$32A^{31}$
XnD22	B^4 out/in	1G				
XnD23		1H				
XnD24		1I				
XnD25		1J				
XnD26			$4E^0$	$8C^0$	$16B^0$	
XnD27			$4E^1$	$8C^1$	$16B^1$	
XnD28			$4F^0$	$8C^2$	$16B^2$	
XnD29			$4F^1$	$8C^3$	$16B^3$	
XnD30			$4F^2$	$8C^4$	$16B^4$	
XnD31			$4F^3$	$8C^5$	$16B^5$	
XnD32			$4E^2$	$8C^6$	$16B^6$	
XnD33			$4E^3$	$8C^7$	$16B^7$	
XnD34		1K				
XnD35		1L				
XnD36		1M		$8D^0$	$16B^8$	
XnD37		1N		$8D^1$	$16B^9$	
XnD38		1O		$8D^2$	$16B^{10}$	
XnD39		1P		$8D^3$	$16B^{11}$	
XnD40				$8D^4$	$16B^{12}$	
XnD41				$8D^5$	$16B^{13}$	
XnD42				$8D^6$	$16B^{14}$	
XnD43				$8D^7$	$16B^{15}$	
XnD49	C^4 in/out					$32A^0$
XnD50	C^3 in/out					$32A^1$
XnD51	C^2 in/out					$32A^2$
XnD52	C^1 in/out					$32A^3$
XnD53	C^0 in/out					$32A^4$
XnD54	C^0 out/in					$32A^5$
XnD55	C^1 out/in					$32A^6$
XnD56	C^2 out/in					$32A^7$
XnD57	C^3 out/in					$32A^8$
XnD58	C^4 out/in					$32A^9$
XnD61	D^4 in/out					$32A^{10}$
XnD62	D^3 in/out					$32A^{11}$
XnD63	D^2 in/out					$32A^{12}$
XnD64	D^1 in/out					$32A^{13}$
XnD65	D^0 in/out					$32A^{14}$
XnD66	D^0 out/in					$32A^{15}$
XnD67	D^1 out/in					$32A^{16}$
XnD68	D^2 out/in					$32A^{17}$
XnD69	D^3 out/in					$32A^{18}$
XnD70	D^4 out/in					$32A^{19}$

43 XS1 库

本章内容：

· 数据类型

· 端口配置函数

· 时钟配置函数

· 端口处理函数

· 时钟处理函数

· 逻辑核心/数据块控制函数

· 信道函数

· 判定函数

· XS1-S 函数

· 辅助函数

43.1 数据类型

clock 时钟源类型。

时钟作为全局变量，使用时钟模块的一个精度标识符启动。运行状态下，时钟向使用该时钟配置的端口提供升降边缘。

43.2 端口配置函数

void configure_in_port_handshake(void port p,

in port readyin,

out port readyout, clock clk)

在握手模式下配置缓冲端口，作为计时的输入端口。

即入或即出端口不是 1 位端口则会引发例外情况。端口缓冲未完成时，通过时钟下降边缘来确定即出端口位置。即入、即出端口位置确定后，端口从时钟取样边缘抽取引脚样本。

默认情况下，端口取样边缘为时钟上升边缘。这可以通过 set_port_sample_delay()函数来改变。

此函数具有以下参数：

p 需配置的缓冲端口

XMOS

readyin 用于即入信号的 1 位端口。

readyout 用于即出信号的 1 位端口。

clk 用于端口配置的时钟。

void configure_out_port_handshake(void port p,

in port readyin,

out port readyout, clock clk,

unsigned initial)

配置缓冲端口，作为握手模式下的计时输出端口。

即入或即出端口不是 1 位端口则会引发例外情况。端口驱动引脚上的初始值，直到输出语句改变驱动值。即入端口可于缓冲端口的取样边缘读取。

自即入端口读取的数值较高时，在时钟下降边缘，如需输出数据，则会将即出端口升高，否则则将即出端口降低。默认情况下，端口取样边为时钟上升边缘。这可以通过函数 set_port_sample_delay()来改变。

此函数具有以下参数：

p 需配置的缓冲端口

readyin 用于即入信号的 1 位端口。

readyout 用于即出信号的 1 位端口。

clk 用于端口配置的时钟。

initial 于端口输出的初始值。

void configure_in_port_strobed_master(void port p,

out port readyout, const clock clk)

配置缓冲端口，作为选通主模式下的计时输入端口。

即出端口不是 1 位端口则会引发例外情况。如端口缓冲未完成时，通过时钟下降边缘来确定即出端口位置。即出端口位置确定后，端口从时钟取样边缘抽取引脚样本。

默认情况下，端口取样边缘为时钟上升边缘。这可以通过 set_port_sample_delay()函数来改变。

此函数具有以下参数：

p 需配置的缓冲端口

XMOS

readyout 用于即出信号的 1 位端口。

clk 用于端口配置的时钟。

void configure_out_port_strobed_master(void port p,

out port readyout,
const clock clk,
unsigned initial)

配置缓冲端口，作为选通主模式下的计时输入端口。

即出端口不是 1 位端口则会引发例外情况。端口驱动引脚上的初始值，直到输出语句改变驱动值。在时钟下降边缘，驱动输出端口。如需输出数据，则会将即出端口升高，否则则将即出端口降低。

此函数具有以下参数：

p 需配置的缓冲端口

readyout 用于即出信号的 1 位端口。

clk 用于端口配置的时钟。

initial 于端口上输出的初始值。

void configure_in_port_strobed_slave(void port p,

in port readyin, clock clk)

配置缓冲端口，作为选通主模式下的计时输入端口。

即入端口不是 1 位端口则会引发例外情况。即入信号较高时，端口从引脚取样边缘取样。默认情况下，端口取样边缘为时钟上升边缘。这可以通过函数 set_port_sample_delay()来改变。

此函数具有以下参数：

p 需配置的缓冲端口

readyin 用于即入信号的 1 位端口。

clk 用于端口配置的时钟。

void configure_out_port_strobed_slave(void port p,

in port readyin, clock clk,

unsigned initial) ·

配置缓冲端口，作为选通主模式下的计时输出端口。

即入端口不是 1 位端口则会引发例外情况。端口驱动引脚上的初始值，直到输出语句改变驱动值。即入端口可于缓冲端口的取样边缘读取。自即入端口读取的数值较高时，在时钟下降边缘，如需输出数据，则会升高即出端口，否则则降低即出端口。默认情况下，端口取样边为时钟上升边缘。这可以通过函数 set_port_sample _delay()来改变。

此函数具有以下参数：

p 需配置的缓冲端口

readyin 用于即入信号的 1 位端口。

clk 用于端口配置的时钟。

initial 于端口上输出的初始值。

void configure_in_port(void port p, const clock clk)

配置端口，作为不含就绪信号的计时输入端口。

这是端口的默认模式，端口自取样边缘取样，如果端口无缓冲，则可通过输出来改变端口方向。这种改变发生在时钟下一下降边缘。此后，端口作为不含就绪信号的输出端口。

默认情况下，端口取样边为时钟上升边缘。这可以通过函数 set_port_sample_delay()来改变。

此函数具有以下参数：

p 需配置的端口，可以为缓冲型也可以为非缓冲型。

clk 用于端口配置的时钟。

void configure_in_port_no_ready(void port p, const clock clk)

为 configure_in_port()的别名。

void configure_out_port(void port p, const clock clk, unsigned initial)

配置端口，作为不含就绪信号的计时输入端口。

端口驱动引脚上的初始值，直到输入或输出语句改变驱动值。于时钟下一下降边缘驱动输出，每个时钟周期需保持一个端口宽度的数据位。如果未缓冲端口，则可通过执行输入来改变端口方向。这种改变发生在输出暂停一个时钟周期之后时钟的下降边缘。此后，端口作为不含就绪信号的输入端口。

此函数具有以下参数：

p 需配置的端口，可以为缓冲型也可为非缓冲型。

clk 用于端口配置的时钟。

initial 于端口上输出的初始值。

void configure_out_port_no_ready(void port p, const clock clk, unsigned initial)

configure_out_port()的别名。

void configure_port_clock_output(void port p, const clock clk)

配置 1 位端口以输出时钟信号。

端口不是 1 位端口则会引发例外情况。在该模式下于端口上执行输入或输出时会产生下列未定义行为。

此函数具有以下参数：

p 需配置的 1 位端口。

clk 用于输出的时钟。

void set_port_no_sample_delay(void port p)

将端口设定为无抽样延迟模式。

使得端口于时钟上升边缘对输入数据取样，这是端口的默认状态。

此函数具有以下参数：

p 需配置的端口。

void set_port_sample_delay(void port p)

将端口设定为取样延迟模式。

这使端口在时钟下降边缘对输入数据进行抽样。

此函数具有以下参数：

p 需配置的端口。

void set_port_clock(void port p, const clock clk)

在端口附加时钟。

这相当于于端口上使用 SETCLK 指令。采用时钟边缘进行数据取样和输出。通常优先使用配置_*_port_* 函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

XMOS

clk 需附加的时钟。

void set_port_ready_src(void port p, void port ready)

设定 1 位端口作为另一即出端口。

这相当于于端口上使用 SETRDY 指令。即出端口不是 1 位端口时则会引发例外情况。在该模式下于端口上执行输入或输出时会产生下列未定义行为。即出端口用于表示端口准备就绪可用于传输数据。通常优先使用配置_*_port_* 函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

ready 1 位端口就绪，可用于传输即出信号。

void set_port_use_on(void port p)

开启端口。

端口状态初始化到此类型端口的默认状态。如果此端口已开启，则将其状态复位到默认态。

此函数具有以下参数：

p 需打开的端口。

void set_port_use_off(void port p)

关闭端口。

端口关闭后不执行任何行动。使用关闭后的端口会引发例外情况。

此函数具有以下参数：

p 需关闭的端口。

void set_port_mode_data(void port p)

配置端口作为数据端口。

这是端口的默认状态；在端口上执行输出操作，以控制输出信号。

此函数具有以下参数：

p 需配置的端口。

void set_port_mode_clock(void port p)

配置 1 位端口作为时钟输出端口。

端口将输出与端口相连的时钟。端口不是 1 位端口则会引发例外情况。可使用 set_port_mode_data() 函数，将端口复位至默认态。

XMOS

此函数具有以下参数：

p 需配置的端口。

void set_port_mode_ready(void port p)

配置 1 位端口用作信号即出端口。

本端口将输出与 set_port_ready_src()连接的即出信号。端口不是 1 位端口则会引发例外情况。可使用 set_port_mode_data()函数将端口复位至默认状态。通常优先使用配置_*_port_*函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

void set_port_drive(void port p)

在驱动模式下配置端口。

输出至端口的数值通过引脚来驱动，这是端口的默认驱动态。调用 set_port_drive()函数，可禁用端口上拉或下拉电阻。

此函数具有以下参数：

p 需配置的端口。

void set_port_drive_low(void port p)

在低驱动模式下配置端口。

1 位端口输出 0 时，会将引脚驱动到较低的位置；输出为 1 时，不会驱动任何值。如果端口不为 1 位端口，则输出至端口的结果未界定。XS1-G 设备调用 set_port_drive_low()时，会同时启用端口内部上拉电阻。XS1-L 设备调用 set_port_drive_low()时，会同时启用端口内部下拉电阻。

此函数具有以下参数：

p 需配置的端口。

void set_port_pull_up(void port p)

启用端口内部上拉电阻。

不驱动引脚时，上拉电阻确保自端口的取样值为 1，然而上拉电阻不足以保证确定的外部值。XS1-G 设备调用 set_port_pull_up()时，会同时在低驱动模式下驱动端口。XS1-L 设备上无上拉电阻时调用 set_port_pull_up()函数，则可能引发例外情况。

此函数具有以下参数：

p 需配置的端口。

void set_port_pull_down(void port p)

启用端口内部下拉电阻。

不驱动引脚时，下拉电阻确保端口取样值为 0，然而下拉电阻不足以保证确定的外部值。XS1-G 设备上无下拉电阻时调用 set_port_pull_down()函数，则可能引发例外情况。XS1-L 设备上调用 set_port_pull_down()函数，则会在低驱动模式下配置端口。

此函数具有以下参数：

p 需配置的端口。

void set_port_pull_none(void port p)

禁用端口的上拉或下拉电阻。

这可以在驱动模式下配置端口。

此函数具有以下参数：

p 需配置的端口。

void set_port_master(void port p)

将端口设定为主模式。

这相当于于值为 XS1_SETC_MS_MASTER 的端口上使用 SETC 指令。通常优先使用 config-ure_in_port_strobed_master()和 configure_out_port_strobed_master()函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

void set_port_slave(void port p)

将端口设定为从属模式。

这相当于于值为 XS1_SETC_MS_SLAVE 的端口上使用 SETC 指令。通常优先使用 config-ure_in_port_strobed_slave() 和 configure_out_port_strobed_slave()函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

void set_port_no_ready(void port p)

配置端口不可使用就绪信号。

这相当于于值为 XS1_SETC_RDY_NOREADY 的端口上使用 SETC 指令。通常优先使用 configure_in_port()和 configure_out_port()函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

void set_port_strobed(void port p)

将端口设定为选通模式。

这相当于于值为 XS1_SETC_RDY_STROBED 的端口上使用 SETC 指令。通常优先使用 _*_port_strobed_*函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

void set_port_handshake(void port p)

将端口设定为握手模式。

这相当于于值为 XS1_SETC_RDY_HANDSHAKE 的端口上使用 SETC 指令。通常优先使用_*_port_handshake 函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

p 需配置的端口。

void set_port_no_inv(void port p)

配置端口，不得转换已经取样且于引脚上驱动的数据。

这是端口的默认状态。

此函数具有以下参数：

p 需配置的端口。

void set_port_inv(void port p)

配置端口，不得转换已经取样且于引脚上驱动的数据。

端口不是 1 位端口则会引发例外情况。如果端口作为时钟源，则设定该模式会交换时钟的上升脉冲和下降边缘。

此函数具有以下参数：

p 需配置的 1 位端口。

XMOS

void set_port_shift_count(void port p, unsigned n)

设置端口的移位计数。

这与 SETPSC 指令相对应，新的移位计数必须小于端口的移位宽度，大于零且大于端口宽度的数倍，否则将引发例外情况。一定量的数据移位时，在端口处输入该函数将使下一输入就绪。捕获数据有效位最多时，下一输入将会返回传输宽度位的数据。在端口处输出该函数将导致下一输出移出相应数位。通常优先使用 partin()和 partout()函数，而非 setpsc()函数，因前两种函数可同时执行所需配置及输入或输出。

此函数具有以下参数：

p 需配置的缓冲端口。

n 新移位计数。

void set_pad_delay(void port p, unsigned n)

在与端口相连的引脚上设定延迟。

该端口引脚上取样的输入信号，于端口上发现之前，会产生一个等同于处理器时钟周期的延迟，引脚上默认延迟为 0，而延迟必须在 0-5 之内设定。如果多个启用端口连接至同一引脚，则通过优先级最高的端口设定该引脚的延迟。

此函数具有以下参数：

p 需配置的端口。

n 输入信号延迟的处理器时钟周期数。

43.3 时钟配置函数

void configure_clock_src(clock clk, void port p)

配置时钟将 1 位端口作为时钟源。

这使得端口上的输入、输出操作与外部时钟信号同步。端口不是 1 位端口则会引发例外情况。

此函数具有以下参数：

clk 需配置的时钟。

p 用作时钟源的 1 位端口。

void configure_clock_ref(clock clk, unsigned char divide)

配置时钟以基准时钟作为时钟源。

如果时钟分频设定为 0，则使用基准时钟频率，否则，将基准时钟频率除以 2 作为分频使用。默认情况下，基准时钟频率配置为 100MHz。

此函数具有以下参数：

clk 需配置的时钟。

divide 时钟分频

void configure_clock_rate(clock clk, unsigned a, unsigned b)

配置时钟以(a/b) MHz 的频率运行。

如果硬件不支持规定频率，则会引发例外情况，硬件支持频率 MHz 和频率形式（MHz 为基准时钟频率，数值为 1-255 之间）。

此函数具有以下参数：

clk 需配置的时钟。

a 所需频率的被除数。

b 所需频率的除数。

void configure_clock_rate_at_least(clock clk, unsigned a, unsigned b)

配置时钟以硬件支持的最低频率即大于等于(a/b) MHz 运行。

频率不符合该要求时会引发例外情况。此函数具有以下参数：

clk 需配置的时钟。

a 所需频率的被除数。

b 所需频率的除数。

void configure_clock_rate_at_most(clock clk, unsigned a, unsigned b)

配置时钟以硬件支持的最快非零频率即大于等于(a/b) MHz 运行。

频率不符合该要求时会引发例外情况。此函数具有以下参数：

clk 需配置的时钟。

a 所需频率的被除数。

b 所需频率的除数。

void set_clock_src(clock clk, void port p)

于 1 位端口上配置时钟源。

这与时钟上使用 SETCLK 指令相对应。端口不是 1 位端口时则会引发例外情况，同时时钟先前设定过非零分频时，也会引发例外情况。通常使用通常优先使用 configure_clock_src()函数，因其不会产生这类问题。

此函数具有以下参数：

clk 需配置的时钟。

p 作为时钟源的 1 位端口。

void set_clock_ref(clock clk)

为基准时钟设计时钟源。

这与时钟上使用 SETCLK 指令相对应，时钟分频保持不变。

此函数具有以下参数：

clk 需配置的时钟。

void set_clock_div(clock clk, unsigned char div)

设计时钟分频。

这与时钟上使用 SETD 指令相对应。必须为基准时钟设计时钟源，否则会引发例外情况。如果时钟分频设定为 0，则源频率保持不变，否则将原频率除以分频的 2 倍。

此函数具有以下参数：

clk 需配置的时钟。

div 使用的分频。

void set_clock_rise_delay(clock clk, unsigned n)

为时钟上升边缘设定延迟。

与时钟相连的端口上发现时钟上升边缘之前，会产生一个等同于处理器时钟周期的延迟。默认上升边缘延迟为 0，而延迟必须在 0-512 之内设定。如果时钟边缘会延迟多个时钟周期，则与时钟相连的端口上无法看到时钟上升边缘。

此函数具有以下参数：

clk 需配置的时钟。

XMOS

n 时钟上升边缘延迟的处理器时钟周期数。

void set_clock_fall_delay(clock clk, unsigned n)

设计时钟下降边缘的延迟。

在与时钟相连的端口上看见时钟下降边缘之前，会产生一个等同于处理器时钟周期的延迟。默认下降边缘延迟为 0，而延迟必须在 0-512 之内设定。如果时钟边缘会延迟多个时钟周期，则与时钟相连的端口上无法看到时钟下降边缘。

此函数具有以下参数：

clk 需配置的时钟。

n 时钟下降边缘延迟的处理器时钟周期数。

void set_clock_ready_src(clock clk, void port ready)

设置时钟将 1 位端口用于即入信号。

这相当于于时钟上使用 SETRDY 指令。端口不是 1 位端口时则会引发例外情况。在该模式下于端口上执行输入或输出时会产生下列未定义行为。即出端口用于表示端口准备就绪可用于传输数据。通常优先使用配置_*_port_* 函数，因其可确保所需模式要求的端口配置按正确顺序变化。

此函数具有以下参数：

clk 需配置的时钟。

ready 用于即入信号的 1 位端口。

void set_clock_on(clock clk)

打开时钟。

时钟状态初始化到此类型时钟的初始状态。如果此时钟开启，则其状态复位到默认态。

此函数具有以下参数：

clk 需打开的时钟。

void set_clock_off(clock clk)

关闭时钟。

时钟关闭后不执行行动。使用关闭后的端口会引发例外情况。

此函数具有以下参数：

clk 需关闭的时钟。

XMOS

43.4 端口处理函数

void start_port(void port p)

启动端口。

清除端口所用的缓冲。

此函数具有以下参数：

p 需启动的端口。

void stop_port(void port p)

停用端口。

端口复位至非就绪端口。
此函数具有以下参数：

p 需停用的端口。

unsigned peek(void port p)

指令端口于引脚上对现值进行取样。

端口立即向处理器提供取样的端口宽度位数据，而不管传输宽度、时钟、就绪信号和缓冲。输入不会对端口上后续输入、输出产生影响。

此函数具有以下参数：

p 达到峰值的端口。

函数返回：

引脚上的取样值。

void clearbuf(void port p)

清除端口使用的缓冲。

未经处理器输入的数据，不得对其取样。由处理器输出但未经端口驱动的数据，不得对其取样。连续输出过程中的端口必须立即终止。如果暂停输出可能改变端口方向，则方向不会改变。如果调用 clearbuf()函数于引脚上驱动数值，则会持续驱动该值直到输出语句改变驱动数值。

此函数具有以下参数：

p 需清除缓冲的端口。

void sync(void port p)

等待直到端口完成所有暂停输出。

等待直到端口暂停输出全部完成，引脚上最后一个端口宽度的数据保持一个时钟周期。

此函数具有以下参数：

p 等候中的端口。

unsigned endin(void port p)

于缓冲端口上终止当前输入。

返回经端口取样但未经处理器输入的位数。该计数包括反序列化所用的传输寄存器和移位寄存器中的数据。端口后续输入返回传输宽度位的数据，直到所剩数据位小于一个传输宽度。只需进一步输入就可以读取剩余数据，当捕获数据有效位最多时，该输入将会返回传输宽度位的数据。

此函数具有以下参数：

p 终止当前输入的端口。

此函数返回：

剩余数据位数。

unsigned partin(void port p, unsigned n)

于缓冲端口上执行规定宽度的数据输入。

宽度必须小于端口的传输宽度，大于零且大于端口宽度的数倍，否则将引发例外情况。端口移位寄存器中位数大于或等于规定宽度值时，返回值未定。

此函数具有以下参数：

p 用于输入的缓冲端口。

n 待输入的数据位数。

此函数返回：

输入值。

void partout(void port p, unsigned n, unsigned val)

于缓冲端口上执行规定宽度的数据输出。

宽度必须小于端口的传输宽度，大于零且大于端口宽度的数倍，否则将引发例外情况。输出最低 n 有效位的数值。

此函数具有以下参数：

XMOS

p 用于输出的缓冲端口。

n 待输出的数据位数。

val 待输出的数值。

unsigned partout_timed(void port p, unsigned n, unsigned val, unsigned t)

端口计数等于规计时间时，于缓冲端口上执行规定宽度的数据输出。

宽度必须小于端口的传输宽度，大于零且大于端口宽度的数倍，否则将引发例外情况。输出最低 n 有效位的数值。

此函数具有以下参数：

p 用于输出的缓冲端口。

n 待输出的数据位数。

val 待输出的数值。

t 待输出的端口计数值。

{unsigned /* value */, unsigned /* timestamp */} p partin_timestamped(void port p,
unsigned n)

于缓冲端口上执行规定宽度的输入，并添加输入时间戳。

宽度必须小于端口的传输宽度，大于零且大于端口宽度的数倍，否则将引发例外情况。端口移位寄存器中位数大于或等于规定宽度值时，返回值未定。

此函数具有以下参数：

p 用于输入的缓冲端口。

n 待输入的数据位数。

函数返回：

输入值及时间戳。

unsigned partout_timestamped(void port p, unsigned n, unsigned val)

于缓冲端口上执行规定宽度的数据输出，并添加输出时间戳。

宽度必须小于端口的传输宽度，大于零且大于端口宽度的数倍，否则将引发例外情况。输出最低 n 有效位的数值。

XMOS

此函数具有以下参数：

p 用于输出的缓冲端口。

n 待输出的数据位数。

val 待输出的数值。

此函数返回：

输出时间戳。

43.5 时钟处理函数

void start_clock(clock clk)

时钟处于运行状态。

时钟只有处于运行状态才会产生边缘。所有与时钟相连的端口计数器都复位为 0。

此函数具有以下参数：

clk 处于运行状态的时钟。

void stop_clock(clock clk)

等待直到时钟较低，而后将其置于停滞状态。

在停滞状态时钟不会产生边缘。

此函数具有以下参数：

clk 将置于停滞状态的时钟。

43.6 逻辑核心/数据块控制函数

void set_core_fast_mode_on(void)

将当前逻辑核心设定至快速运行模式。

排程器为快速运行模式下的逻辑核心预留一个槽，而不管核心是等待输入还是等待完成选择。这就降低了暂停输入或选择完成时状态变化所带来的最坏情况延迟。然而，将核心置于最快模式意味着核心处于等待状态时，其他逻辑核心都不能使用附加槽。除此之外，将逻辑核心设定为快速运行模式还可能增加耗电量。

void set_core_fast_mode_off(void)

将当前逻辑核心设定至正常执行模式。

XMOS

如果先前将核心设定为快速模式，则使用 set_core_fast_mode_on()函数将执行模式恢复至默认状态。

unsigned getps(unsigned reg)

获取处理器状态寄存器值。

这与 GETPS 指令相对应。如果参数不是合法处理器的状态寄存器，则会引发例外情况。

此函数具有以下参数：

reg 需读取的处理器状态寄存器。

此函数返回：

处理器状态寄存器值。

void setps(unsigned reg, unsigned value)

设定处理器状态寄存器值。

这与 GETPS 指令相对应。如果不是合法处理器状态寄存器，则会引发例外情况。

此函数具有以下参数：

reg 需写入的处理器状态寄存器。

value 处理器状态寄存器设定值。

int read_pswitch_reg(unsigned tileid, unsigned reg, unsigned &data)

读取处理器开关寄存器值。

该读数为规定 编号的处理器开关读数。成功时会返回 1，此时为数据赋予寄存器值。收到故障应答，或寄存器值或数据块标识符太大而与读取包不匹配时，返回 0.

此函数具有以下参数：

tileid 数据块标识符。

reg 寄存器号。

data 自寄存器读取数值。

此函数返回：

读数是否成功。

int write_pswitch_reg(unsigned tileid, unsigned reg, unsigned data)

向处理器开关寄存器写入数值。

该写入为规定 编号的处理器开关写入值。成功时会返回 1，此时

向数据分配寄存器值。收到故障应答，或寄存器值或数据块标识符太大而与写入包不匹配时，返回 0。

此函数具有以下参数：

tileid 数据块标识符。

reg 寄存器号。

data 写入寄存器的数值。

此函数返回：

写入是否成功。

int write_pswitch_reg_no_ack(unsigned tileid, unsigned reg, unsigned data)

向处理器开关寄存器写入值而无需确认。

该写入为规定编号的处理器开关读数。与 write_pswitch_reg()不同，该函数无需一直等候写入执行。寄存器值或数据块标识符太大而与写入不匹配时，返回 0，否则返回 1。未对故障确认作出规定，因此返回值反映不出写入是否成功。

此函数具有以下参数：

tileid 数据块标识符。

reg 寄存器号。

data 写入寄存器的值。

此函数返回：

参数是否有效。

int read_sswitch_reg(unsigned tileid, unsigned reg, unsigned &data)

读取系统开关寄存器值。

该读数为规定 编号的处理器开关读数。成功时会返回 1，此时为数据赋予寄存器值。收到故障应答，或寄存器值或数据块标识符太大而与读取包不匹配时，返回 0。

此函数具有以下参数：

tileid 数据块标识符。

reg 寄存器号。

data 写入寄存器的值。

此函数返回：

XMOS

读数是否成功。

int write_sswitch_reg(unsigned tileid, unsigned reg, unsigned data)

向系统开关寄存器写入值。

该写入是向规定编号的系统开关写入。收到成功应答返回 1，收到故障应答，或寄存器值或数据块标识符太大而与写入包不匹配时，返回 0。

此函数具有以下参数：

tileid 数据块标识符。

reg 寄存器号。

data 写入寄存器的值。

此函数返回：

写入是否成功。

int write_sswitch_reg_no_ack(unsigned tileid, unsigned reg, unsigned data)

向系统开关寄存器写入值，无需确认。

该写入是向规定编号的系统开关写入。收到成功应答返回 1，收到故障应答，或寄存器值或数据块标识符太大而与写入包不匹配时，返回 0，否则返回 1。由于无需确认，返回值不能反映是否成功。

此函数具有以下参数：

tileid 数据块标识符。

reg 寄存器号。

data 写入寄存器的值。

此函数返回：

参数是否有效。

int read_node_config_reg(tileref tile, unsigned reg, unsigned &data)

读取节点配置寄存器值。

读取含规定数据块的节点值，读取成功时会返回 1 且为数据赋予寄存器值。如果收到故障确认或寄存器值太大而与读取包不匹配时，返回 0。

此函数具有以下参数：

tile 数据块。

XMOS

reg 寄存器号码。

data 自寄存器读取的数值。

此函数返回：

读取是否成功。

int write_node_config_reg(tileref tile, unsigned reg, unsigned data)

向节点配置寄存器写入值。

写入含规定数据块的节点值，读取成功时会返回 1。如果收到故障确认或寄存器值太大而与写入包不匹配时，返回 0。

此函数具有以下参数：

tile 数据块。

reg 寄存器号码。

data 向寄存器写入值。

此函数返回：

读取是否成功。

int write_node_config_reg_no_ack(tileref tile, unsigned reg, unsigned data)

向节点配置寄存器写入值，无需确认。

写入含规定数据块的节点值，与 write_node_config_reg() 不同，此函数在写入执行之前无需等待。寄存器值太大而与写入包不匹配时返回 0，否则返回 1。由于不需确认，因此返回值不能反映写入是否成功。

此函数具有以下参数：

tile 数据块。

reg 寄存器号码。

data 向寄存器写入值。

此函数返回：

参数是否有效。

int read_periph_8(tileref tile,

unsigned peripheral,

unsigned base_address,
unsigned size,

unsigned char data[])

从以指定基地址为起点的外围处读取字节。

外围必须为带 8 位界面的外围，成功时返回 1，以读取到的数值填充数据，失败时返回 0。

此函数具有以下参数：

tile 数据块。

peripheral 外围号码。

base_address 基地址。

size 需读取的 8 位数值。

data 自外围读取的数值。

此函数返回：

读取是否成功。

int write_periph_8(tileref tile,

unsigned peripheral,

unsigned base_address,

unsigned size,

const unsigned char data[])

向以指定基地址为起点的外围处写入字节。

外围必须为带 8 位界面的外围，成功时返回 1，失败时返回 0。

此函数具有以下参数：

tile 数据块。

peripheral 外围号码。

base_address 基地址。

size 需写入的 8 位数值。

data 向外围写入的数值。

此函数返回：

写入是否成功

int write_periph_8_no_ack(tileref tile,

unsigned peripheral,

unsigned base_address, unsigned size,

const unsigned char data[])

向以指定基地址为起点的外围处写入字节，无需确认。

外围必须为带 8 位界面的外围，与 write_periph_8()不同，此函数与写入执行之前无需等待。由于不需确认，因此返回值不能反映写入是否成功。

此函数具有以下参数：

tile 数据块。

peripheral 外围号码。

base_address 基地址。

size 需写入的 8 位数值。

data 向外围写入的数值。

此函数返回：

参数是否有效。

int read_periph_32(tileref tile,

unsigned peripheral,

unsigned base_address,
unsigned size,

unsigned data[])

自以指定基地址为起点的外围处读取 32 位字节。

外围必须为带 8 位界面的外围，成功时返回 1，以读取到的数值填充数据，失败时返回 0。读取带 8 位界面的外围时，各字有效字节为最小地址的字节（大端字节顺序）。

此函数具有以下参数：

tile 数据块。

peripheral 外围号码。

base_address 基地址。

XMOS

size 需读取的 32 位字符数。

data 自外围读取的数值。

此函数返回：

读取是否成功。

int write_periph_32(tileref tile,

unsigned peripheral,

unsigned base_address,
unsigned size,

const unsigned data[])

向以指定基地址为起点的外围处写入 32 位字节。

成功时返回 1，失败时返回 0。向带 8 位界面的外围写入数据时，将传递给该函数的最有效字节写入最小地址的字节（大端字节顺序）。

此函数具有以下参数：

tile 数据块。

peripheral 外围号码。

base_address 基地址。

size 需写入的 32 位字符数。

data 向外围写入的数值。

此函数返回：

写入是否成功。

int write_periph_32_no_ack(tileref tile,

unsigned peripheral,

unsigned base_address, unsigned size,

const unsigned data[])

向以指定基地址为起点的外围处写入 32 位字节，无需确认。

与 write_periph_32()不同，此函数与写入执行之前无需等待。由于不需确认，因此返回值不能反映写入是否成功。向带 8 位界面的外围写入数据时，将传递给该函数的最有效字节写入最小地址的字节（大端字节顺序）。

此函数具有以下参数：

tile 数据块。

peripheral 外围号码。

base_address 基地址。

size 需写入的 32 位字符数。

data 向外围写入的数值。

此函数返回：

参数是否有效。

unsigned get_local_tile_id(void)

返回调用器所运行的数据块标识符。

标识符唯一鉴别网络中的数据块。

此函数返回：

数据块标识符。

unsigned get_logical_core_id(void)

返回调用器所运行的逻辑核心标识符。

标识符唯一鉴别当前数据块上的逻辑核心。

此函数返回：

逻辑核心标识符。

43.7 信道函数

void start_streaming_master(chanend c)

启动信道上的流式通讯。

调用该函数时必须同时调用信道另一端的 start_streaming_slave()函数。开通两个信道端点的路径，以使用流式输入、输出函数进行非同步通讯。使用 stop_streaming_master()函数关闭之前，本路径一直保持开启状态。注意，如果网络中两点之间开启的信道数等于这两点之间所有可能的路径数，则这两点之前不存在其他信道通讯，这将使程序陷入死循环。

此函数具有以下参数：

c 数据流开始的信道端

void stop_streaming_master(chanend c)

终止信道中的流式通讯。

调用该函数时必须同时调用信道另一端的 stop_streaming_slave()函数。关闭先前通过 start_streaming_master()函数开启的路径，使路径可用于其他信道通讯。

此函数具有以下参数：

c 数据流终止的信道端

void start_streaming_slave(chanend c)

启动信道中的流式通讯。

调用该函数时必须同时调用信道另一端的 start_streaming_master()函数。开通两个信道端点的路径，以使用流式输入、输出函数进行非同步通讯。使用 stop_streaming_slave()函数关闭之前，本路径一直保持开启状态。注意，如果网络中两点之间开启的信道数等于这两点之间所有可能的路径数，则这两点之前不存在其他信道通讯，这将使程序陷入死循环。选择语句下可调用 start_streaming_slave()函数，使信道另一端调用 start_streaming_master()函数后，此函数处于就绪状态。

此函数具有以下参数：

c 数据流开始的信道端。

void stop_streaming_slave(chanend c)

终止信道中的流式通讯。

调用该函数时必须同时调用信道另一端的 stop_streaming_master()函数。关闭先前通过 start_streaming_slave()函数开启的路径，使路径可用于其他信道通讯。

此函数具有以下参数：

c 数据流终止的信道端

void outuchar(chanend c, unsigned char val)

流式输出一个值作为信道端的无符号字符。

所使用的协议与输入(:>)和输出(<:)操作符使用的协议不兼容。

此函数具有以下参数：

c 流式输出数据的目的信道端。

val 输出的数值。

void outuint(chanend c, unsigned val)

流式输出一个值作为信道端的无符号字符。

所使用的协议与输入(:>)和输出(<:)操作符使用的协议不兼容。

此函数具有以下参数：

c 流式输出数据的目的信道端。

val 输出的数值。

unsigned char inuchar(chanend c)

自信道端流式输入无符号字符。

如果信道中的下一符号为控制符，则会引发例外情况。所使用的协议与输入(:>)和输出(<:)操作符使用的协议不兼容。

此函数具有以下参数：

c 流式输入数据的目的信道端。

此函数返回：

收到的值。

unsigned inuint(chanend c)

自信道端流式输入无符号字符。

如果信道中的下一字节包含控制符，则会引发例外情况。所使用的协议与输入(:>)和输出(<:)操作符使用的协议不兼容。

此函数具有以下参数：

c 流式输入数据的目的信道端。

此函数返回：

收到的值。

void inuchar_byref(chanend c, unsigned char &val)

自信道端流式输入无符号字符。

向 val 写入输入值。如果信道中的下一符号为控制符，则会引发例外情况。所使用的协议与输入(:>)和输出(<:)操作符使用的协议不兼容。

此函数具有以下参数：

c 流式输入数据的目的信道端。

val 设定接收值的变量。

void inuint_byref(chanend c, unsigned &val)

自信道端流式输入无符号整数。

向 val 写入输入值。选择语句下可调用 start_streaming_slave()函数，在这种语句只要数据可用于信道时，则此函数就处于就绪状态。所使用的协议与输入(:>)和输出(<:)操作符使用

的协议不兼容。

此函数具有以下参数：

c 流式输入数据的目的信道端。

val 设定接收值的变量。

void outct(chanend c, unsigned char val)

于信道端流式输出控制符。

试图输出硬件控制符将引发意外情况。

此函数具有以下参数：

c 流式输出数据的目的信道端。

val 将输出的控制符值。

void chkct(chanend c, unsigned char val)

检查规定值的控制符。

信道中下一字节为控制符，且与期望值相符，则输入该控制符而后删除，否则将引发例外情况。

此函数具有以下参数：

c 信道端。

val 期望控制符值。

unsigned char inct(chanend c)

于信道端流式输入控制符。

信道中下一字节不为控制符时，则会引发例外情况，否则返回控制符值。

此函数具有以下参数：

c 流式输入数据的信道端。

此函数返回

接收到的控制符。

void inct_byref(chanend c, unsigned char &val)

于信道端流式输入控制符。

向 val 写入输入值。信道中下一字节不为控制符时，则会引发例外情况

此函数具有以下参数：

c 流式输入数据的信道端。

val 设定接收值的变量。

XMOS

int testct(chanend c)

测试信道端下一字节是否为控制符。

不从信道中删除控制符，因此控制符仍可用于输入。

此函数具有以下参数：

c 执行测试的信道端。

此函数返回：

下一字节为控制符返回 1，否则返回 0。

int testwct(chanend c)

测试信道端下一字节是否包含控制符。

字符中包含控制符，则返回字符中控制符的位置。不从信道中删除任何数据。

此函数具有以下参数：

c 执行测试的信道端。

此函数返回：

字符中首个控制符的位置（1-4）；字符中不包含控制符时返回 0。

void soutct(streaming chanend c, unsigned char val)

于流式信道端输出控制符。

试图输出硬件控制符将会引发例外情况。试图输出控制符 CT_END 或 CT_PAUSE 是无效的行为。

此函数具有以下参数：

c 流式输出数据的目的信道端。
val 输出的控制符值。
void schkct(streaming chanend c, unsigned char val)
于流式信道端检查给定值的控制符。
信道中下一字节为控制符，且与期望值相符，则输入该控制符而后删除，否则将引发例外情况。
此函数具有以下参数：
c 流式信道端。
val 期望控制符值。
unsigned char sinct(streaming chanend c)
于流式信道端输入控制符。
信道中下一字节不为控制符时，则会引发例外情况，否则返回控制符值。
此函数具有以下参数：

c 流式输入数据的信道端。
此函数返回：
收到的控制符。
void sinct_byref(streaming chanend c, unsigned char &val)
于流式信道端输入控制符。
向 val 写入输入值。如果信道中的下一符号不为控制符，则会引发例外情况。
此函数具有以下参数：
c 流式输入数据的流式信道端。
val 设定接收值的变量。
int stestct(streaming chanend c)
测试信道端下一字节是否为控制符。
不从信道中删除控制符，因此控制符仍可用于输入。
此函数具有以下参数：
c 执行测试的信道端。
此函数返回：
下一字节为控制符返回 1，否则返回 0.
int stestwct(streaming chanend c)
测试信道端下一字节是否包含控制符。

字符中包含控制符，则返回字符中控制符的位置。不从信道中删除任何数据。

此函数具有以下参数：

c 执行测试的流式信道端。

此函数返回：

字符中首个控制符的位置（1-4）；字符中不包含控制符时返回 0。

transaction out_char_array(chanend c, const char src[], unsigned size)

通过信道输出数据块。

于信道端输出数据的全部字节。调用 out_char_array()函数时必须同时调用信道另一端的 in_char_array()函数。输出字节数必须与输入字节数匹配。

此函数具有以下参数：

c 用于输出的信道端。

src 发送的值数组。

size 输出的字节数。

transaction in_char_array(chanend c, char src[], unsigned size)

自信道输入数据块。

于信道端输入数据的全部字节并存入数组中。调用 in_char_array() 函数时必须同时调用信道另一端的 out_char_array()函数。输入字节数必须与输出字节数匹配。

此函数具有以下参数：

c 用于输入的信道端。

src 存放自信道输入数值的数组。

size 输入字节数。

43.8 判定函数

void pinseq(unsigned val)

等待直到端口引脚数值与规定值相等。

必须调用该函数作为端口上输入的表达式。端口引脚上的数值与 val 端口宽度的有效位数相等时，输入就绪。

此函数具有以下参数：

val 相比对的数值。

void pinsneq(unsigned val)

等待直到端口引脚数值与规定值不相等。

必须调用该函数作为端口上输入的表达式。端口引脚上的数值与 val 端口宽度的有效位数相等时，输入就绪。

此函数具有以下参数：

val 相比对的数值。

void pinseq_at(unsigned val, unsigned time)

等待直到端口引脚数值与规定值相等，且端口计数与规计时间相等。

必须调用该函数作为非缓冲端口上输入的表达式。端口引脚上的数值与 val 端口宽度的有效位数相等且端口计数器与规计时间相等时，输入就绪。

此函数具有以下参数：

val 相比对的数值。

time 比对的时间

void pinsneq_at(unsigned val, unsigned time)

等待直到端口引脚数值与规定值不相等，而端口计数器与规计时间相等。

XMOS

必须调用该函数作为非缓冲端口上输入的表达式。端口引脚上的数值与 val 端口宽度的有效位数相等且端口计数器与规计时间时，输入就绪。

此函数具有以下参数：

val 相比对的数值。

time 比对的时间

void timerafter(unsigned val)

等待直到计时器时间与规定值相等。

必须调用该函数作为计时器上输入的表达式。计时器计数通过晚于规定值即晚于给定值来解释后，输入就绪。表达式为真时，时间 A 即视为晚于时间 B。

此函数具有以下参数：

val 相比对的时间。

43.9 XS1-S 函数

此函数用于控制 XS1-S 设备上的模拟数字转换器。

void enable_xs1_su_adc_input(unsigned number, chanend c)

启用由数字规定的模拟数字转换器输入。

将样本送至信道端 C。

此函数具有以下参数：

number 模拟数字转换器输入号码。

c 与 XS1-SU 模拟数字转换器相连的信道。

void enable_xs1_su_adc_input_streaming(unsigned number, streaming chanend c)

启用由数字规定的模拟数字转换器输入。

将样本送至信道端 C。

此函数具有以下参数：

number 模拟数字转换器输入号码。

c 与 XS1-SU 模拟数字转换器相连的信道。

void disable_xs1_su_adc_input(unsigned number, chanend c)

禁用由数字规定的模拟数字转换器输入。

此函数具有以下参数：

number 模拟数字转换器输入号码。

c 与 XS1-SU 模拟数字转换器相连的信道。

void disable_xs1_su_adc_input_streaming(unsigned number, streaming chanend c)

禁用由数字规定的模拟数字转换器输入。

此函数具有以下参数：

number 模拟数字转换器输入号码。

c 与 XS1-SU 模拟数字转换器相连的信道。

43.10 辅助函数

void crc32(unsigned &checksum, unsigned data, unsigned poly)

于循环冗余检查和中加入字节。

执行下列计算：

```
for (int i = 0; i < 32; i++) {
  int xorBit = (crc & 1);

  checksum  = (checksum >> 1) | ((data & 1) << 31);
  data = data >> 1;

  if (xorBit)
    checksum = checksum ^ poly;
}
```

此函数具有以下参数：

checksum 检查和的初始值，随最新检查和一起更新。

data 循环冗余检查的数据。

poly 循环冗余检查计算时所用的多项式。

unsigned crc8shr(unsigned &checksum, unsigned data, unsigned poly)

于循环冗余检查和内并入 8 位字节。

于 8 位以上有效数据位上计算循环冗余检查，数据右移 8 位返回。执行的计算如下：

```
for (int i = 0; i < 8; i++) {
  int xorBit = (crc & 1);

  checksum  = (checksum >> 1) | ((data & 1) << 31);
  data = data >> 1;

  if (xorBit)
    checksum = checksum ^ poly;
}
```

此函数具有以下参数：

checksum 检查和的初始值，随最新检查和一起更新。

data 数据

poly 循环冗余检查计算时所用的多项式。

此函数返回：

数据右移 8 位。

{unsigned, unsigned} l lmul(unsigned a, unsigned b, unsigned c, unsigned d)

乘以 2 个字节生成一个双字，并加入两个单字。

返回结果的高字和低字。此乘法无符号，不能超值。执行的计算如下：

```
(uint64_t)a * (uint64_t)b + (uint64_t)c + (uint64_t)d
```

此函数返回：

计算出的高值和低值。

{unsigned, unsigned} m mac(unsigned a, unsigned b, unsigned c, unsigned d)

乘以 2 个字节生成一个双字，并加入一个双字。
返回结果的高字和低字。执行的计算如下：

此函数返回：

XMOS

```
(uint64_t)a * (uint64_t)b + (uint64_t)c<<32 + (uint64_t)d
```

计算出的高值和低值。

{signed, unsigned} m macs(signed a, signed b, signed c, unsigned d)

乘以 2 个带符号字节并添加一个双字，生成一个双字。

返回结果的高字和低字。执行的计算如下：

```
(int64_t)a * (int64_t)b + (int64_t)c<<32 + (int64_t)d
```

此函数返回：

计算出的高值和低值。

signed sext(unsigned a, unsigned b)

符号扩展输入。

第一个参数为符号扩展值，第二个参数包含一个数位位置。将较高或等同位置的所有数位值设置为较低一位的值。实际上，较低的数位 b 通过带符号的整数来解释。如果 b 小于 1 或大于 32，则该结果与参数 a 相等。

此函数返回：

符号扩展值。

unsigned zext(unsigned a, unsigned b)

零扩展输入。

第一个参数为零扩展值，第二个参数包含一个数位位置。将较高或等同位置的所有数位值设置为 0。实际上，较低的 b 数位通过不带符号的整数来解释。如果 b 小于 1 或大于 32，则该结果与参数 a 相等。

此函数返回：

零扩展值。

XMOS

44 xCORE 32 位应用二进制界面

工具开发指南[8]中包含了 XS1 32 位应用二进制界面、XE 文件格式和系统调用界面。

8 http://www.xmos.com/docnum/X9114

第 O 部分平台配置

内容

· 描述目的平台
· XN 规范

45 描述目的平台

本章内容

· 支持的网络拓扑
· 带两个程序包的板

通过 XN 描述硬件平台。XN 文件向 XMOS 编译器工具链提供关于目的硬件如 XMOS 设备、端口、闪存和振荡器的资料。

XMOS 工具使用 XN 数据产生名为<platform.h>规定平台的头文件，编译、启动并调试多节点程序。

45.1 支持的网络拓扑

工具支持下列网络拓扑。

	网络拓扑	支持的配置
图 81：支持的网络拓扑	线形	仅适用 XS1-L 设备，最多 16 个节点
	超立方	2 级（两个节点） 3 级（4 个节点一圈） 3 级（8 个节点一个立方） 4 级（16 个节点典型立方）

45.2 带两个程序包的板

图 82 描述了带两个成一直线 XMOS L1 设备的板，下文描述了一个相匹配的 XN。

XN 文件以 XML 声明开始。

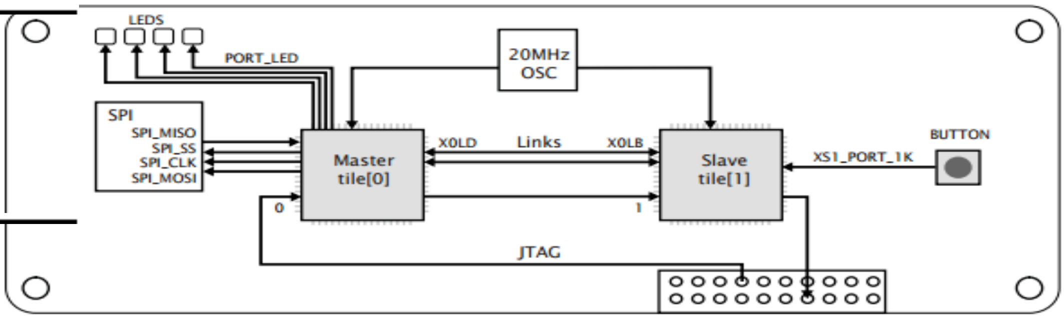
```
<?xml version="1.0" encoding="UTF-8"?>
```

下列代码可启动网络。

```
<Network xmlns="http://www.xmos.com"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.xmos.com http://www.xmos.com">
```


XMOS

图 82 示例硬件平台



下列代码描述了两个 xCORE 数据块，声明“tileref tile[2];”输出至名为<platform.h>的头文件。

```
<Declarations>
  <Declaration>tileref tile[2]</Declaration>
</Declarations>
```

下列代码描述了名为 P1 的程序包，包括一个名为 Master 的单节点。

```
<Packages>
  <Package Id="P1" Type="XS1-L1A-TQ128">
    <Nodes>
      <Node Id="Master" Type="XS1-L1A" InPackageId="0"
            Oscillator="20MHz" SystemFrequency="400MHz">
      <Boot>
        <Source Location="SPI:bootFlash"/>
        <Bootee NodeId="Slave" Tile="0"/>
      </Boot>
      <Tile Number="0" Reference="tile[0]">
        <Port Location="XS1_PORT_1A" Name="PORT_SPI_MISO"/>
        <Port Location="XS1_PORT_1B" Name="PORT_SPI_SS"/>
        <Port Location="XS1_PORT_1C" Name="PORT_SPI_CLK"/>
        <Port Location="XS1_PORT_1D" Name="PORT_SPI_MOSI"/>
        <Port Location="XS1_PORT_4A" Name="PORT_LED"/>
      </Tile>
    </Node>
  </Nodes>
</Package>
```

Master 节点为 TQ128 程序包中的一个 400MHz XS1-L1A 设备，通过 20MHz 振荡器计时。从名为“bootFlash”的 SPI 设备中启动，该 SPI 设备等级为“SPIFlash”。

数据块“0”的说明与数据块【0】相关，而端口 1A、1B、1C、 1D 和 4A 为规定符号名称，这些说明输出至名为<platform.h>的头文件。

下列代码描述了名为 P2 的程序包，包括一个名为 Slave 的单节点。

```
  <Package Id="P2" Type="XS1-L1A-TQ128">
    <Nodes>
      <Node Id="Slave" Type="XS1-L1A" InPackageId="0"
            Oscillator="20Mhz" SystemFrequency="400MHz">
        <Boot>
          <Source Location="LINK"/>
        </Boot>
        <Tile Number="0" Reference="tile[1]">
          <Port Location="XS1_PORT_1K" Name="PORT_BUTTON"/>
        </Tile>
      </Node>
    </Nodes>
  </Package>
</Packages>
```

Slave 节点为 TQ128 程序包中的一个 400MHz XS1-L1A 设备，通过 20MHz 振荡器计时。从 Master 节点处通过 xConnect Link 启动。

下面代码定义了一个 2 线 xConnect 链路，在 Master 节点上与链路 X0LD 连接，在 Slave 节

```
<Links>
  <Link Encoding="2wire" Delays="4,4">
    <LinkEndpoint NodeId="Master" Link="X0LD"/>
    <LinkEndpoint NodeId="Slave" Link="X0LB"/>
  </Link>
</Links>
```

点上与链路 X0LB 连接。

链路符号内和符号间的迟延为 4 个时钟周期。

下列编码描述了与 XMOS 设备相连板上的部件清单。

```
<ExternalDevices>
  <Device NodeId="Master" Tile="0" Name="bootFlash"
          Class="SPIFlash" Type="AT25FS010">
     <Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO"/>
     <Attribute Name="PORT_SPI_SS"   Value="PORT_SPI_SS"/>
     <Attribute Name="PORT_SPI_CLK"  Value="PORT_SPI_CLK"/>
     <Attribute Name="PORT_SPI_MOSI" Value="PORT_SPI_MOSI"/>
   </Device>
 </ExternalDevices>
```

名为 bootFlash 设备于节点 Master 上与 xCORE 数据块 0 连接，并赋予属性——将设备上四个 SPI 引脚与端口连接（通过 XFLASH 识别级别 XFLASH）。

下列代码描述了 JTAG 扫描链。

```
  <JTAGChain>
    <JTAGDevice NodeId="Master" Position="0"/>
    <JTAGDevice NodeId="Slave" Position="1"/>
  </JTAGChain>

</Network>
```

46 XN 规范

本章内容：

- 网络单元
- 声明
- 程序包
- 节点
- 链路
- 设备
- JTAGDevice

46.1 网络单元

xTIMEcomposer 支持单一 XN 文件，该文件包含一个单一网络定义，此网络定义如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<Network xmlns="http://www.xmos.com"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.xmos.com http://www.xmos.com">
```

图 62 中列出了网络单元的 XN 层次。

46.2 声明

声明单元为一个或多个 xCORE 数据块提供符号名称。单一名称或多个名称通过下列形式支持：

tileref 标识符

tileref 标识符【常量表达】

同等声明输出至名为<platform.h>的头文件，用于 XC 程序。Tileref 声明通过数据块单元（见§42.4.1）参考属性，与 xCORE 物理数据块相关。

示例

```
<Declaration>tileref master</Declaration>
<Declaration>tileref tile[8]</Declaration>
```

节点	编号	描述	章节

图 83： 单元的 XN 层次

网络	1	xCORE 网络	
声明	0+		
声明	1+	xCORE 数据块声明	§??
程序包	1+		
程序包	1+	设备程序包	§??
节点	1		
节点	1+	节点声明	§??
数据块	1+	xCORE 数据块	§??
端口	0+	xCORE 符号型端口名	§??
启动	0 或 1	启动方式	§??
信号源	1	二进制定位	§??
启动对象	0+	启动的节点	§??
服务	0+	服务声明	§??
信道端	1+	信道端参数	§??
链路	0 或 1		
链路	1+	xConnect 链路声明	§??
链路端点	2	xConnect 链路端点	§??
外部设备	0 或 1		
设备	1+	外部设备	§??
属性	0+	设备属性	§??
JTAG 链	0 或 1		
JTAG 设备	1+	JTAG 链中的设备	§??

46.3 程序包

程序包单元指一个程序包文件，描述来自 xCORE 端口和链路，至程序包引脚上的映射。

图 63：XN 程序包	属性	必要性	类型	描述
	标识符	必要	字串	程序包名称，网络中所有程序包名称必须是惟一的。
	类型	必要	字串	XML 程序包名称，于 XCC_DEVICE_PATH 规定路径中搜索.pkg 类型文件的工具。

示例

文件 XS1-L2A-QF124.xml 中描述了名为 L2 的程序包。

```
<Package id="L2" Type="XS1-L2A-QF124">
```


XMOS

46.4 节点

节点单元描述了网络中一组 xCORE 数据块，所有数据块与单个开关相连。XMOS 设备如 G4 或 L1 都为节点示例。

属性	必要性	类型	描述
标识符	不必要	字串	节点名称，网络中所有节点名称必须是唯一的。
类型	必要	字串	如果类型为 periph:XS1-SU，则节点为 XS1-SU 外围节点。否则，类型则为描述节点的 XML 文件名。于 XCC_DEVICE_PATH 规定路径中搜索.pkg 类型文件的工具。
基准	必要	字串	将节点与声明中规定的 xCORE 数据块标识符相连。该属性适用于与类型为 periph:XS1-SU 的节点。
InPackageId	必要	字串	将节点映射至程序包文件中的单元。
振荡器	不必要	字串	PLL 振荡器输入频率通过数字加单位 MHz、KHz 或 Hz 的形式规定。
OscillatorSrc	不必要	字串	提供 PLL 振荡器输入的节点名称。
SystemFrequency	不必要	字串	系统频率通过数字加单位 MHz、KHz 或 Hz 的形式规定，未设置时默认为 400MHz。
PllFeedbackDivMin	不必要	整数	最小容许的 PLL 反馈分频器，未设置时默认为 1.
ReferenceFrequency	不必要	字串	基准时钟频率，通过数字加单位 MHz、KHz 或 Hz 的形式规定，未设置时默认为 100MHz
PllDividerStageOneReg	不必要	整数	PLL 分频器 1 级寄存器值。
PllMultiplierStageReg	不必要	整数	PLL 倍增器 1 级寄存器值。
PllDividerStageTwoReg	不必要	整数	PLL 分频器 2 级寄存器值。
RefDiv	不必要	整数	SystemFrequency / RefDiv =ReferenceFrequency

图 85：XN 节点

可使用 SystemFrequency、PllFeedbackDivMin 和 ReferenceFrequency 属性自动配置 PLL 寄存器，或使用 PllDividerStageOneReg、PllMultiplierStageReg、PllDividerStageTwoReg 和 RefDiv 手动配置寄存器。如果提供了前三个属性，则无需提供后四个属性，反之亦然。

PLL 振荡器输入频率可通过 Oscillator 或 OscillatorSrc 属性规定，如果提供了 Oscillator 属性，则无需提供 OscillatorSrc 属性，反之亦然。

使用手动配置时，必须提供 PllDividerStageOneReg、PllMultiplierStageReg、PllDividerStageTwoReg 和 RefDiv 属性，此时必须规定 PLL 振荡器的输入频率。工具使用这些值来设定 PLL 寄存器和基准时钟分频器。可于 XS1-G 处理器（见 X3221）和 XS1-L 处理器（见 X1433）的 xCORE 频率控制文档中查看 PLL 分频器上的信息。

如果振荡器频率已定且未提供手动 PLL 属性，则使用自动配置。工具对 PLL 寄存器进行编程，从而达到目的系统频率，PLL 反馈分频器大于等于最小值，此时达到目的基准时钟频率。如果任何一种限制条件不满足，则工具会发出警报并报告实际使用值。

如果未规定振荡器频率，则工具不会对 PLL 进行配置，节点引脚所确定的 PLL 寄存器

初始值保持不变。

一个网络可以含 XS1-L 设备或 XS1-G 设备中的任意一个，但不同时拥有两个。

示例

```
<Node Id="MyL1" Type="XS1-L1A" Oscillator="20Mhz"
      SystemFrequency="410MHz" ReferenceFrequency="98.5Mhz">
```

如文件 config_XS1-L1A.xml 所述，名为 MyL1 的节点是 L1 设备。

46.4.1 数据块

数据块单元描述了单个 xCORE 数据块的性质。

属性	必要性	类型	描述
号码	必要	整数	节点中数据块的唯一号码，必须为 0 至 n-1 之间的任意值，其中，n 为节点 XML 文件中的数据块数。
基准	不必要	字串	将数据块与声明中 tile[n]形式的标识符相关联。一个数据块最多能与一个标识符相关联。

图 86 XN Tileref 单元

示例

```
<Tile Number="0" Reference="tile[0]">
```

46.4.2 端口

端口单元提供了端口的符号型名称。

示例

属性	必要性	类型	描述
位置	必要	字串	名为<xs1.h>标准头文件中定义的端口标识符。XC 操作手册（见 X1009）中描述了这些端口。
名称	必要	字串	有效的 C 处理器标识符；网络中说明的所有端口名称必须是惟一的。

图 87：XN 端口单元

```
<Port Location="XS1_PORT_1I" Name="PORT_UART_TX"/>
<Port Location="XS1_PORT_1J" Name="PORT_UART_RX"/>
```

46.4.3 启动

Boot 单元定义了节点是如何启动的，它包含一个“源”单元（见§42.4.4）及零或更多个从 xConnect 链路启动的 Bootee 单元（见§42.4.5）。如果 Source 单元规定了 xConnect 链路，则无需规定 Bootee 单元。在 XS1-L 设备路线中，启动对象可根据自 SPI 启动的设备进行配置。

XMOS 工具要求 Boot 单元可以从闪存中启动程序（见§22.1）。

46.4.4 源

Source 单元规定了节点启动的位置，具有以下属性。

图 88：XN 源单元	属性	必要性	类型	描述
	定位	必要	字串	形式为 SPI 或 LINK，一组设备单元中包含了设备名称。

只能配置 XMOS XS1-L 设备，通过 xConnect 链路启动。

示例

```
<Source Location="SPI:bootFlash"/>
```

46.4.5 启动对象

Bootee 单元陈述了系统中通过 xConnect 链路启动的另一节点。如果该节点与其一个启动对象（见§42.5 和§42.5.1）之间的 xConnect 链路超过一条，则工具会选择其中的一条用于启动。

图 89：XN Bootee 单元	属性	必要性	类型	描述
	节点标识符	必要	字串	可用于另一节点的标识符

示例

```
<Bootee NodeId="Slave">
```

46.4.6 比特

Bit 单元陈述了首个不匹配数位与特定位号相符的信息方向，

图 90：XN Bit 单元	属性	必要性	类型	描述
	位号	必要	整数	比特位，自最低有效位起编号
	方向	必要	整数	信息路径方向

示例

```
<Bit number =" 1 "  direction =" 0 " / >
```

46.4.7 链接

位于路由表中的链接单元规定了 xCONNECT 链路的方向。

图 91：XN Link 单元	属性	必要性	类型	描述
	名称	必要	字符串	连接标识为 XnLm 型，其中 n 表示数据块数，而 m 为链接号。关于可用链接引出请看相应的包装数据表。
	方向	必要	整数	链路方向

46.4.8 服务

Service 单元规定了节点提供的 XC 服务函数。

图 92：XN Service 单元	属性	必要性	类型	描述
	原型	必要	字串	服务函数原型，不包括服务关键字。本原型输出至名为<platform.h>的头文件，用于 XC 程序。

示例

```
<Service Proto="service_function(chanend c1, chanend c2)">
```

46.4.9 信道端

Chanend 单元描述了 XC 服务函数所用的信道端参数。

图 93 ： XN Service 单元	属性	必要性	类型	描述
	标识符	必要	字串	服务函数原型中信道端参数识别器
	端点	必要	整数	当前节点上的信道端号码。
	远程	必要	整数	与当前节点信道端相连的远程信道端号码。

示例

```
<Chanend Identifier="c" end="23" remote="5"/>
```


XMOS

46.5 链路

系统规范文档（XS1- G: X7507，XS1-L: X1151）和链路性能文档（XS1-G: X7561, XS1-L: X2999）中描述了 xConnect 链路。

Link 单元描述了 xConnect 链路的特征，必须确切包含两个 LinkEndpoint 产物（见§46.5.1）。

图 94：XN 链路单元	属性	必要性	类型	描述
	编码	必要	字串	必须为两线或五线
	延迟	必要	字串	为 x、y 型，其中，x 规定了端点的外部迟延，y 规定了端点的内部迟延。省略 y 值，则使用 x，1；两值皆省略，则使用 1.
	旗标	必要	字串	规定了链路的附加属性，使用 XSCOPE 值来确定使用的链路，从而向 XScope 发送追踪信息。

示例

```
<Link Encoding="2wire" Delays="4,4">
```

46.5.1 LinkEndpoint

LinkEndpoint 描述了 xConnect 链路的一个端点，详情见系统规范文档（XS1-G: X7507、XS1-L: X1151）。每个端点将节点标识符与物理 xConnect 链路相关联。

图 95 ： XN LinkEndpoint 单元	属性	必要性	类型	描述
	节点标识符	不必要	字串	有效的节点标识符。
	链路	不必要	字串	链路标识符为 XnLm 型，其中，n 代表数据块号，m 为链接信。关于可用链路引脚分配，请看相应程序包数据表。
	路由标识符	不必要	整数	xConnect 链路网络上的路由标识符。
	信道端	不必要	整数	信道端

端点通常为节点标识符与链路标识符的组合。在流式调试链路中，任意一个端点都必须为路由标识符与信道端的组合。

示例

```
<LinkEndpoint NodeId="0" Link="X0LD"/>
<LinkEndpoint RoutingId="0x8000" Chanend="1">
```


XMOS

46.6 设备

Device 单元描述与 xCORE 数据块相连但不直接与 xConnect 链路相连的设备

图 96 : XN Device 单元	属性	必要性	类型	描述
	名称	必要	字串	对设备命名的标识符。
	节点标识符	必要	字串	与设备相连节点的标识符。
	数据块	必要	整数	与设备相连节点的数据块。
	等级	必要	字串	设备等级
	类型	不必要	字串	设备类型（与等级相关）

xTIMEcomposer 识别 SPIFlash 级，并使用类型属性来区分闪存设备型号。

46.6.1 属性

属性单元描述设备的某一方面特质（见§42.6)。

图 74 : XN Attribute 单元	属性	必要性	类型	描述
	名称	必要	字串	规定设备的属性
	值	必要	字串	规定与属性相关的值

XTIMEcomposer 支持 SPIFlash 级别的下列设备属性名称：

PORT_SPI_MISO

SPI 主进从出信号

PORT_SPI_SS

SPISlave 选择信号

PORT_SPI_CLK

SPI 时钟信号

PORT_SPI_MOSI

SPI 主出从进信号

示例

```
<Attribute Name="PORT_SPI_MISO" Value="PORT_SPI_MISO"/>
```

46.7 JTAGDevice

xTIMEcomposer 使用 JTAG 于目的硬件上加载、调试程序。JTAGChain 单元描述 JTAG 链路中的设备。这些单元的顺序决定了它们于 JTAG 链路中的顺序。

图 98： XN JTAGDevice 单元	属性	必要性	类型	描述
	节点标识符	必要	字串	有效的节点标识符

示例

```
<!-- N1 comes before N2 in the JTAG chain -->
<JTAGDevice NodeId="N1">
<JTAGDevice NodeId="N2">
```