SIEMENS

SINUMERIK 840D sl

编程手册 12/2012

SINUMERIK Operate
编程包

制造商文档

SIEMENS

SINUMERIK 840D sl

SINUMERIK Operate
编程包

编程手册

适用于

软件版本 4.5 SP2

12/2012

SINUMERIK®文档

文档标识

以下是当前版本及以前各版本的简要说明。

每个版本的状态由“注释”栏中的代码标识。

在“注释”栏中的状态码分别表示：

A .... 新文档。
B .... 没有改动，但以新的订货号重印。
C .... 经过修订的新版本。

版本	订货号	注释
12/2012	-----	**A**

商标

所有带有所有权保护符号®的都是西门子股份公司的注册商标。该印刷品中的其他名称也可能是商标，任何第三方擅自使用此商标将会侵犯注册商标所有人的权利。

免责条款

我们已对印刷品中所述内容与硬件和软件的一致性作过检查。 尽管如此，仍然不能排除有偏差之处，因此我们不承担保证完全一致的责任。本印刷品中的有关信息会定期审核，而且一些必要的修改会包含在下一个版本中。

订货号-----

目录

1

1 引言

1.1 系统架构和软件架构

图 1 向您展示了 SINUMERIK Operate 软件是如何集成到 SINUMERIK sl 控制系统中的。其中，HMI 与 NCK、PLC 和驱动器之间的通讯是通过软总线实现的，该总线的功能相当于 PLC 的 K 总线。软总线使用的是 S7 通讯协议。外部接入的 HMI 通过通讯处理器(CP)来访问系统内的 NCK、PLC 和驱动器。同理，系统内部的 HMI 也可以通过 CP 来访问其他 NCU 内的 NCK、PLC 和驱动器。
NCK 软件和由 Linux 操作系统控制的部件在同一个架构为 x86 的处理器中执行。而 PLC 软件和驱动器软件则分别在单独的处理器中执行。
操作界面的可视化（输入/输出）是由一个所谓的薄型客户单元（Thin-Client-Unit，简称 TCU）实现的，在图 1-1 中未加以显示。TCU 实际上是一个单独的操作设备，通过以太网与控制设备(NCU)进行通讯。图 1-1 中标为“TCU 接入组件”的组件包含了 TCU 运行所需的服务器（即 VNC 服务器）和 TCU 在 SINUMERIK 系统环境下运行所需的其他必要功能，比如有：多个 TCU 在一个 NCU 的 HMI 上运行时操作焦点的管理功能、TCU 上相连 USB 设备的访问逻辑等。

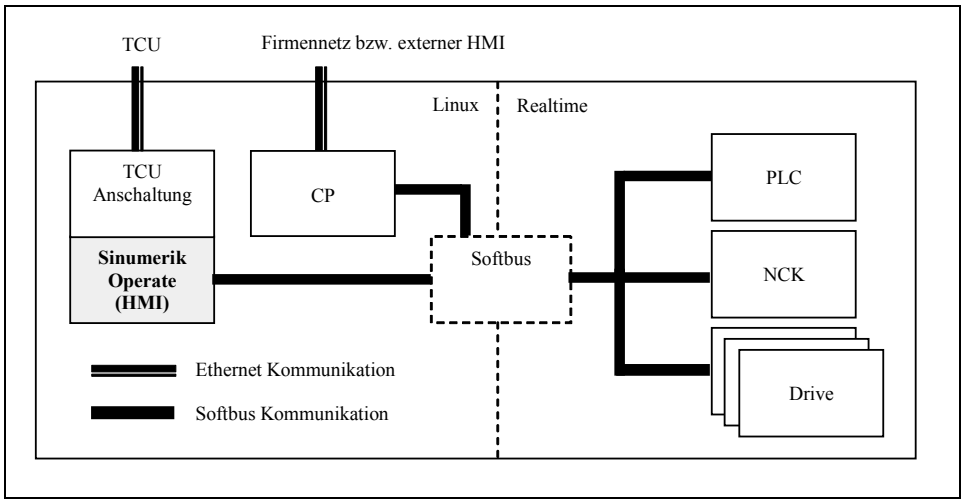


图 1-1:SINUMERIK Operate 软件集成到 SINUMERIK sl 控制系统中的方式

SINUMERIK Operate 具有一种以组件为导向的架构。组件可以分为两大类：GUI 组件和 HMI 服务。其中，GUI 组件是图形化的用户界面，是 SINUMERIK Operate 中用于实现系统与用户即时互动的组件(HMI-GUI)。典型的 GUI 组件有：各个 HMI 话框与各大操作区域（加工、操作、程序、程序管理、诊断和调试）。HMI 服务则用于实现基本功能以及将系统集成到自动化过程中，基本功能囊括了所有 GUI 组件都

需要使用的功能，比如：对和语言相关文本的管理和对数据存储器的访问等功能。典型的 HMI 服务有：读写控制变量的服务；为 HMI 提供所有待处理报警和零件程序信息的报警服务和事件服务；对 NC、CF 卡以及其他驱动器（例如网络盘或 U 盘）上的文件和目录进行简单操作的文件服务。
目前 OA 只能以 GUI 组件的形式进行扩展。SINUMERIK Operate 编程包尚不支持 HMI 服务的开发。

GUI 组件和 HMI 服务可以灵活组合，构建您需要的 HMI 系统。GUI 组件及其需要使用的 HMI 服务要根据所需的 HMI 功能进行配置。只有经过配置后，GUI 组件以及 HMI 服务才会在程序运行时加载到存储器中，然后被执行。此外，GUI 组件以及 HMI 服务既可以在同一个进程中执行，也可以在分布在多个进程中执行。HMI 组件在各个进程中的分布情况对于组件自己而言是完全一目了然的，也就是说，一个 GUI 组件看不到自己使用的 HMI 服务是在同一个进程中执行还是在另一个进程中执行。不管哪种配置，对 HMI 服务的访问都是通过一个接口实现的，而是始终是同一个接口。

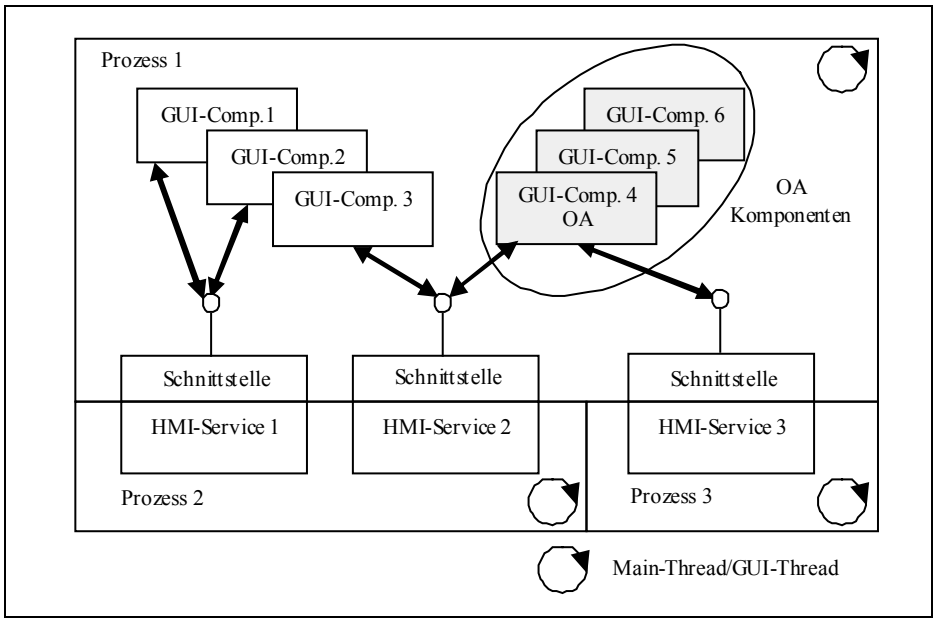


图 1-2:HMI 配置

图 1-2 向您展示了一个 HMI 配置范例，其中有六个 GUI 组件和三个 HMI 服务，它们分布在三个处理器上执行。在这六个 GUI 组件中，GUI 组件 4 到 GUI 组件 6 是 OA 扩展组件，比如：额外增加的操作区域。GUI 组件 1 到 GUI 组件 6 统一在进程 1 中执行。而 HMI 服务 1 和 HMI 服务 2 在进程 2 中执行，HMI 服务 3 在进程 3 中执行。关于具体的 HMI 配置使用了哪些进程、有哪些 GUI 组件和 HMI 服务以及这些 GUI 组件和 HMI 服务在哪个进程中执行的相关信息在一份系统配置文件中加以说明。该文件在 HMI 启动时被读入系统管理器中。系统管理器启动文件中指出的进程，将文件中一同指出的 GUI 组件和 HMI 服务加载到这些进程中。
每个进程至少包含了一个线程，即所谓的主线程（见图 1-2），下文也称 GUI 线程。该线程在生成进程时由系统自动生成。GUI 组件通常在该主线程中执行。
SINUMERIK Operate 是一个由事件控制的系统，也就是说：一旦出现特定事件，比如：某个软键被按下，系统便调用 GUI 组件中的特定函数，然后在主线程中执行这些调用的函数。 在编写 GUI 组件时要注意避免主线程被一些繁重的耗时任务阻塞，

这种阻塞会减慢整个系统的响应速度，HMI 在此期间无法对用户的输入作出任何响应。GUI 组件可能会执行一些比较耗时的任务时，最好在该 GUI 组件中创建辅助线程(Workerthread)，然后在该辅助线程中执行耗时任务。
HMI 服务可在其接口中提供同步函数和异步函数。同步函数只有在完成所有任务后才返回到调用者。在函数返回前，调用者的线程一直被挂起，以避免调用一些耗时函数时执行其他事件。函数返回到调用者后，其结果便可使用。和同步函数相反，异步函数只是触发特定任务的执行，它会立即返回到调用者。函数的真正结果随后通过一个回调函数或一个 Qt 信号提供给调用者。调用在主线程中执行的 HMI 服务时，最好始终采用异步函数，这样可以避免耗时函数阻塞主线程的执行。

! **重要提示**

机床关键 OA 功能必须通过适合的实时应用来加以保护。此类机床关键功能的实现绝对不能只基于 HMI 功能。尤其不得将“Hotlink 机制”（数值修改通知）作为实现机床关键功能的唯一基础。

1.2 平台通用性和开发工具

SINUMERIK Operate 可支持操作系统平台 Windows 和嵌入式 Linux，也就是说：SINUMERIK Operate 源文件是平台通用的，源文件然后借助 Windows 和 Linux 专用的编译程序与链接程序分别转化为各系统可执行的文件。SINUMERIK Operate 源文件的这种平台通用性是通过工具套件“Qt”实现的。Qt 是一个专用于开发简易 GUI 应用程序的工具套件。Qt 的核心是一个 C++类库，它将 Windows 和 Linux 专用的 API 压缩在一起，替换为平台通用的接口。Qt 为每个支持的操作系统提供了一个单独的平台通用接口：Qt/Windows 和 Qt/Embedded(Qtopia)。
Qt/Windows 主要基于 Windows 窗口系统以及 Windows GDI 以输出图形基元。
Qt/Embedded 具有单独的窗口系统，直接使用 Linux 帧缓冲。

<table>
<tr><td colspan="2">Plattformneutrale
HMI sl Software</td></tr>
<tr><td colspan="2">Plattformneutrales Qt API</td></tr>
<tr><td>Qt/Windows
Implementierung</td><td>Qt/Embedded
Implementierung</td></tr>
<tr><td>Windows GDI</td><td>Linux Framebuffer</td></tr>
<tr><td>Windows</td><td>Linux</td></tr>
</table>

图 1-3:平台通用的 SINUMERIK Operate 软件

除了功能丰富的类库外，Qt 还提供一系列可支持 GUI 应用程序开发的工具，比如：Qt Designer，它可以用类似于 Visual Basic 的方式设计出互动式的图形化用户界面。

2 SINUMERIK Operate Windows

2

2.1 创建一个SINUMERIK Operate项目

2.1.1 安装Visual Studio向导

点击SINUMERIK Operate编程包中的“.setup”文件，即可自动安装该软件。安装时文件自动保存到Visual Studio的安装路径下。在重启Visual Studio后即可立即选择和运行向导。

注

按照以下路径启动Visual Studio：
开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → 工具 → Visual Studio

随后Visual Studio利用SINUMERIK Operate专有的环境变量启动。缺少这些环境变量有些功能无法使用，比如：“Custom Build Step”。

2.1.2 打开Visual Studio向导

打开Visual Studio并点击“New Project”弹出新建项目对话框后，“Visual C++ Projects”节点下出现了一个新的文件夹“HMI”。 对话框右侧显示“HMI Project”模板。选中模板后，新项目首先自动命名为SlOem1。如果选中的文件夹中有同名项目，名称末尾的数字会自动调高。当然，您也可以自行修改项目名称。
向导在所有后续步骤中生成的名称（比如：类名称和文件名称等）都以此项目名称为准。

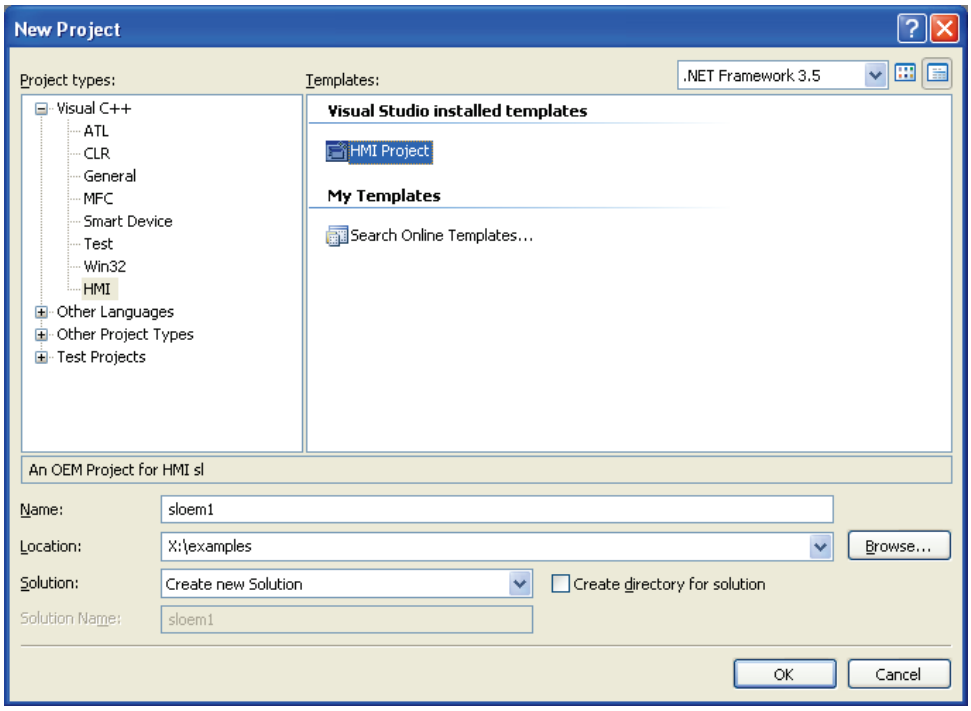


图2-1:选择项目

点击“OK”确认后，真正的SINUMERIK Operate向导开始启动。它可以引导您完成项目的配置，创建项目。

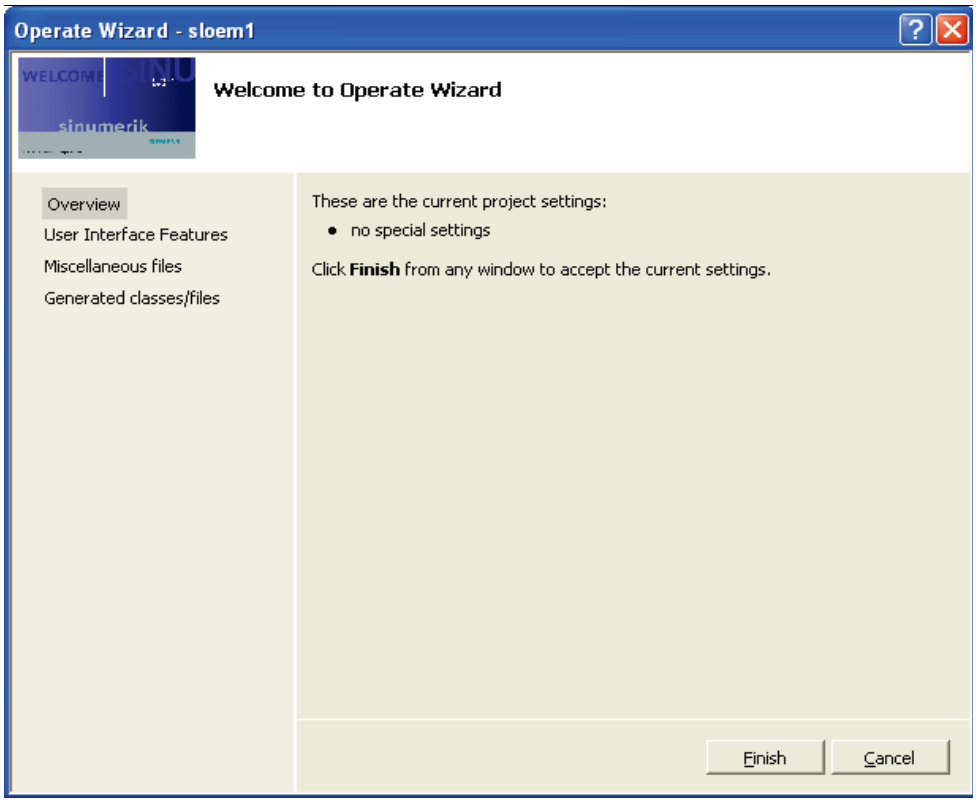


图2-2:配置项目

2.1.3 操作Visual Studio向导

向导界面分为四大模块：

- Overview
- User Interface Features
- Miscellaneous files
- Generated classes/files

Overview

“Overview”中列出了您在向导中完成的所有设置。
一开始时项目没有任何设置，如图2-2所示。如果此时您关闭向导，项目只有基本配置。它既没有任何类，也没有任何配置文件或语言支持文件。项目只有三份文件：

- slexversion.cpp：版本文件
- .CPP：插件定义文件
- .XML：对话框定义文件

只要您在其他模块中进行了设置，这些设置就会显示在“Overview”中，以便您随时查看完成的项目设置。

示例：

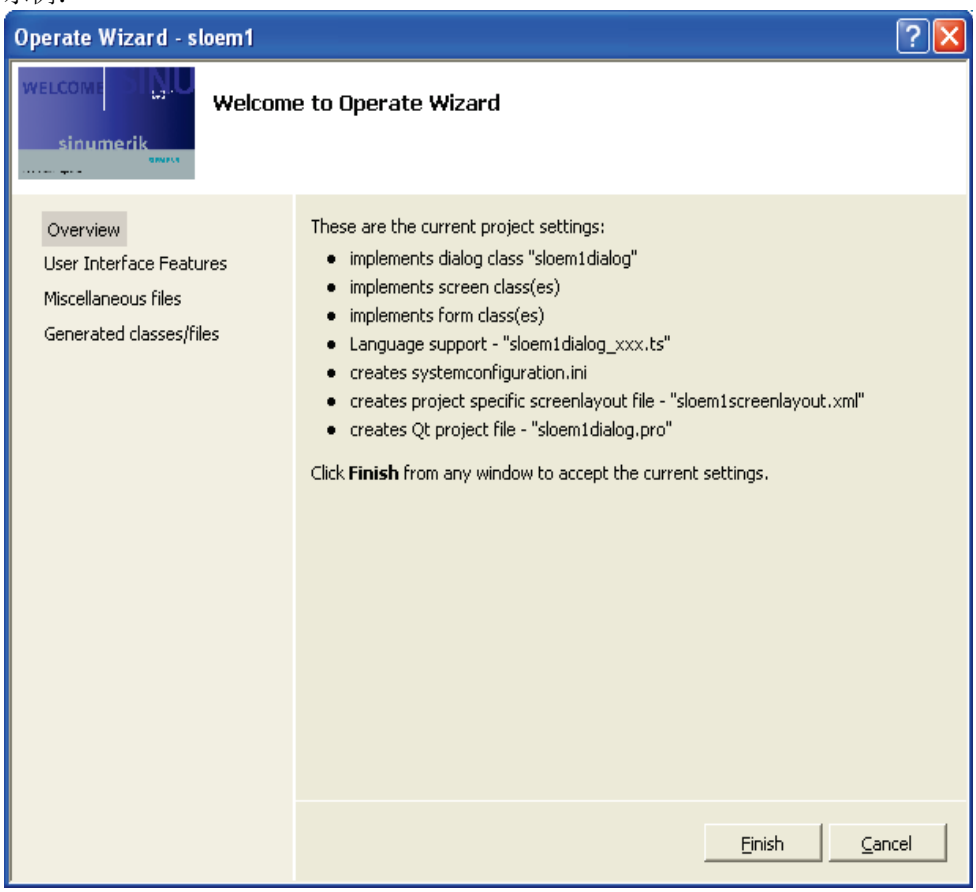


图2-3:Overview

User Interface Features

在本模块中，您可以生成对话框类（dialog class）、屏幕类（screen class）和窗体类（form class）、激活语言支持并修改一些文件的名称。

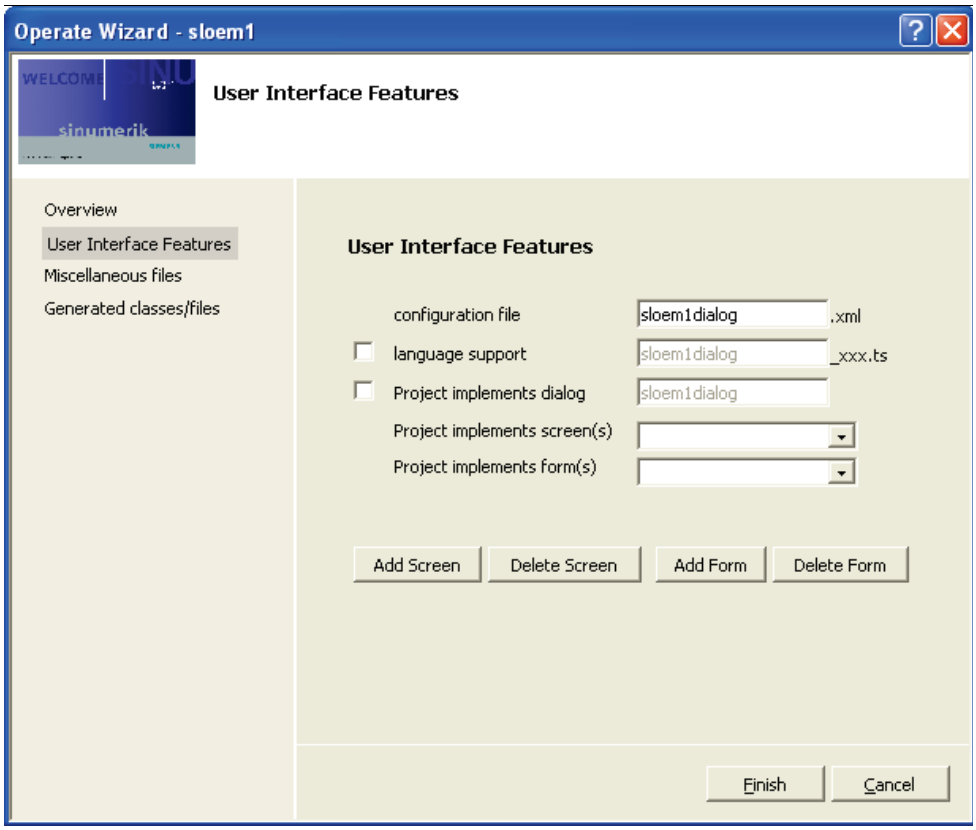


图2-4:User Interface Features

Configuration file

第一栏是配置文件栏，原则上每个SINUMERIK Operat向导项目中都会生成该文件。因此，这一栏前面没有设置复选框，您无法将它删除。但是您可以编辑该文件的名称。

Language support

第二栏是语言支持栏。
如果您勾选了该栏，在生成项目时向导会在项目文件夹下创建一个名为“/languages”的文件夹，其中有六份语言文件，用以下语言标识加以区分：

- DEU（表示德语）
- CHS（表示中文）
- ENG（表示英文）
- ESP（表示西班牙语）
- FRA（表示法语）
- ITA（表示意大利语）

这些语言文件的内容都一样，目的只在于向您展示如何创建和语言相关的文本。

Project implements dialog
勾选了该栏后，项目会包含一个对话框类。向导会依据项目名称自动为该对话框类命名。当然，您也可以自行命名。
Project implements Screen(s)/Form(s)
这两栏用于创建多个屏幕和多个窗体。缺省设置下这两个下拉菜单为空，也就是说：当前没有向项目添加任何屏幕和窗体。
需要向项目添加一个屏幕时，按下按钮“Add Screen”。随后另一个对话框弹出，您可以在其中修改缺省的屏幕名称。
在您确认修改后，向导返回到“User Interface Features”。现在在屏幕的下拉菜单中出现了刚刚创建的屏幕。按照此步骤，您可以创建多个屏幕。
窗体的创建方式如此类似，只是在按下按钮“Add Form”后，除了可以修改窗体名称外您还可以创建一份Ui文件，该文件用于在Qt Desginer中修改窗体。缺省设置中该文件未被选中。
需要删除一个屏幕类或窗体类时，在下拉菜单中选择该类，然后按下按钮“Delete Screen”或“Delete Form”。该类立即从下拉菜单中删除，不添加到项目中。

Miscellaneous files

第三个模块为您提供项目的另外三个设置选项。

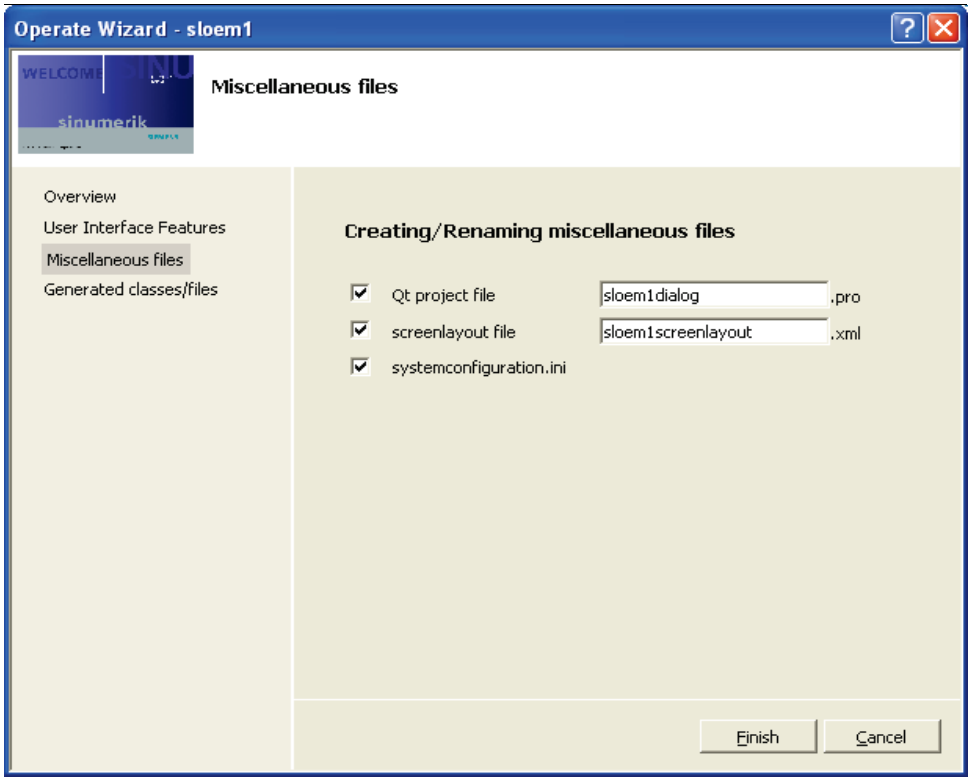


图2-5:Miscellaneous files

Qt project files
扩展名为“.pro”的项目文件用于在Linux系统下生成项目。
勾选该栏后，您可以修改.pro文件的名称。缺省名称是向导根据项目名称提供的。

screenlayout file
该栏用于创建一份Screenlayout文件，以便自定义本项目的屏幕布局。
勾选该栏后，您可以修改Screenlayout文件的名称。缺省名称是向导根据项目名称提供的。

systemconfiguration.ini
勾选该栏后，向导会生成一份本项目的systemconfiguration.ini文件。您可以立即使用该文件，只是该文件不包含标准操作区，只包含已建项目上的显示区域。

Generated classes/files

至少创建了一个类（对话框类、屏幕类或窗体类）后（类的创建参见：2.1.3 -> User Interface Features -> Project implements Screen(s)/Form(s)），才可选择该模块。否则会显示出错。

至少有一个类时，向导显示以下一览：

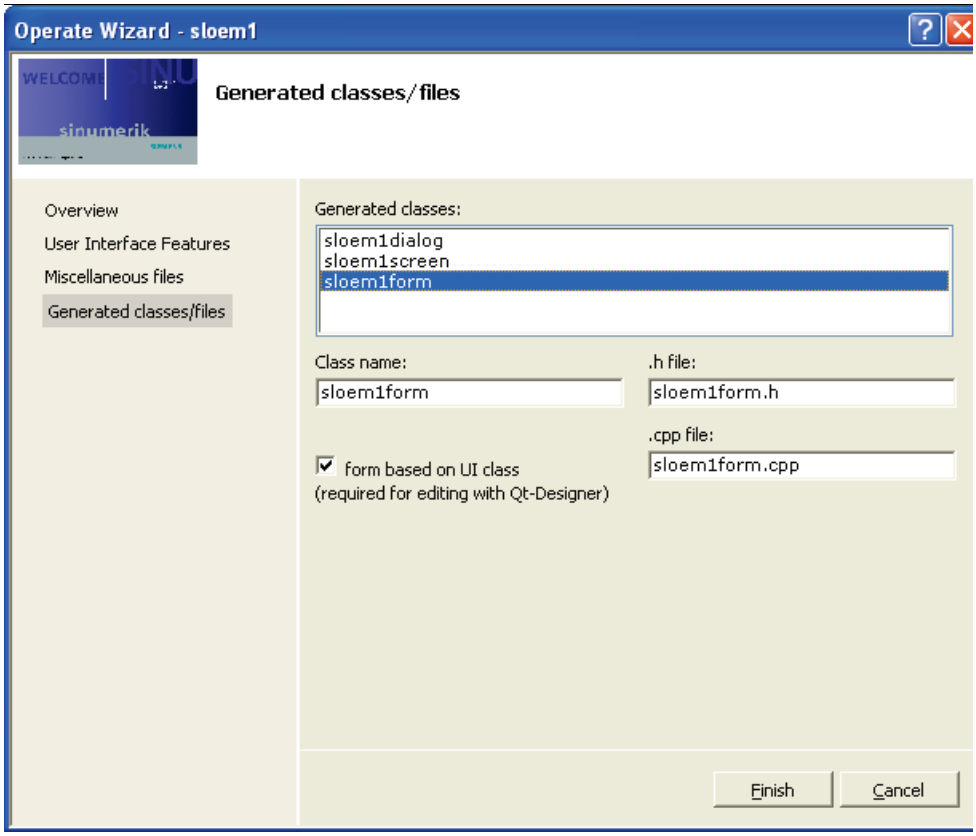


图2-6:Generated classes/files

Generated classes
下拉菜单中列出了迄今为止创建的所有对话框类、屏幕类和窗体类。选中一个类后，它的属性会显示在下拉菜单下方的编辑栏内：

- 类别名称
- 头文件的名称
- CPP文件的名称

- 和Ui文件的关联性（选项）

您可以随意修改所有类名称，只要这些名称符合类或文件的命名规范并且在当前项目中还没有使用过。

Class name
该编辑栏指出了选中类的名称。

.h-file
该编辑栏指出了选中类的头文件的名称。

.cpp-file
该编辑栏指出了选中类的执行文件的名称。

注

在命名.h文件和.cpp文件时，最好统一用小写。这是因为在处理文件名称的大小写时Windows和嵌入式LINUX有所不同。

Form based on UI class
该选项只有在选中类是一个窗体类时才显示。
勾选该选项后，您可以在Qt Designer中编辑窗体。您可以在此修改该选项。

其它

和在其他所有Visual Studio的标准向导中一样，SINUMERIK Operate向导的所有模块中也都包含了一个“Finish”按钮，点击该按钮可以立即开始创建项目。

2.1.4 创建项目时生成的文件

生成的文件及其内容

缺省设置
每个由向导创建的项目都带有一份XML文件和一份插件文件。该XML文件包含了初步的对话框配置。原则上每个窗体都有一个预定义的软键。该配置文件可以由编译程序编译。在插件文件中，每个创建的类都有对应的导出宏命令。
此外，项目创建还会生成以下内部项目文件夹：

表2-1：项目文件夹

项目文件夹	描述
Source files	包含了项目的源代码文件(*.cpp)
Header files	包含了项目的头文件(*.h)
Resource files	包含了所有XML文件（屏幕布局文件和配置文件）以及语言支持文件(*.ts)
Generated	包含了MOC-CPP文件、UI头文件以及生成的其他文件
Forms	包含了项目的窗体(*.ui)

类文件
每个在向导中创建的类（对话框、屏幕或窗体）都带有一份头文件和一份CPP文件。类已经过执行，包含了构造函数和析构函数。所有类都可以保持原样地立即生成。

UI文件/MOC文件
如果为某个窗体类选择了支持Qt Designer编辑，向导会在项目文件夹“Forms”下生成一份Ui文件(*.ui)。
用Qt Designer打开该文件后，您会看到一个空窗体，窗体一个缺省标签，用窗体名称标注。
为支持Qt Designer中的编辑，文件夹“Generated”中生成了对应的Moc文件和Ui头文件，但是只有在编译后，它们才保存到存储器中。换句话说，在第一次开始编译前，您是无法打开这两份文件的。

语言支持
如果您创建的项目设置了语言支持，向导会在项目文件夹“Resource Files”中生成六份语言文件，每份代表不同的语言：德语、英语、法语、西班牙语、意大利语和简体中文。所有文件的内容都相同，都包含了一个范例，向您展示如何进行翻译。

屏幕布局文件
另一份可选择性创建的文件是屏幕布局文件（.xml）。该文件用于针对不同的分辨率定义屏幕布局。

Systemconfiguration.ini
除了内部项目文件夹外，如果您在向导中勾选了“systemconfiguration.ini”文件，向导也会生成该文件。将该文件复制到文件夹“\hmisl\oem\sinumerik\hmi\cfg”下，HMI下一次启动时该应用程序便会随系统一起启动。

.pro文件
如果您在向导中勾选了对应的选项，向导也会在项目中生成一份.pro文件。该文件列出了可以在Qt Designer中编辑的窗体的所有源文件、头文件和Ui文件。
此外，在缺省设置中，Linux系统中生成项目所需的路径和缺省设置都包含在该文件中。

Visual Studio的设置

除了生成各文件外，您还可对Visual Studi项目进行下述设置。

Custom Build Step
所有头文件自动设置Custom Build Step用于Moc’ing。所有Ui文件都可通过Custom Build Step由用户界面编译器UIC编译。
此外，在编译项目时，XML文件和语言文件由编译程序编译。

路径
SINUMERIK Operate编程包中包括文件(*.h)和库文件(*.lib)的路径被添加到项目中。缺省设置中由向导生成的项目为“调试版”。但是您也可以自行生成用于PCU50的“释放版”。

2.1.5 扩展一个SINUMERIK Operate项目

插入对话框类、屏幕类或窗体类

要在现有SINUMERIK Operate项目中插入一个对话框类、屏幕类或窗体类时，可点击Visual Studio中的以下菜单项：

菜单 Project → Add Class…

现在节点“Visual C++”下出现了新文件夹“HMI”。您可以在右侧选择所需类：

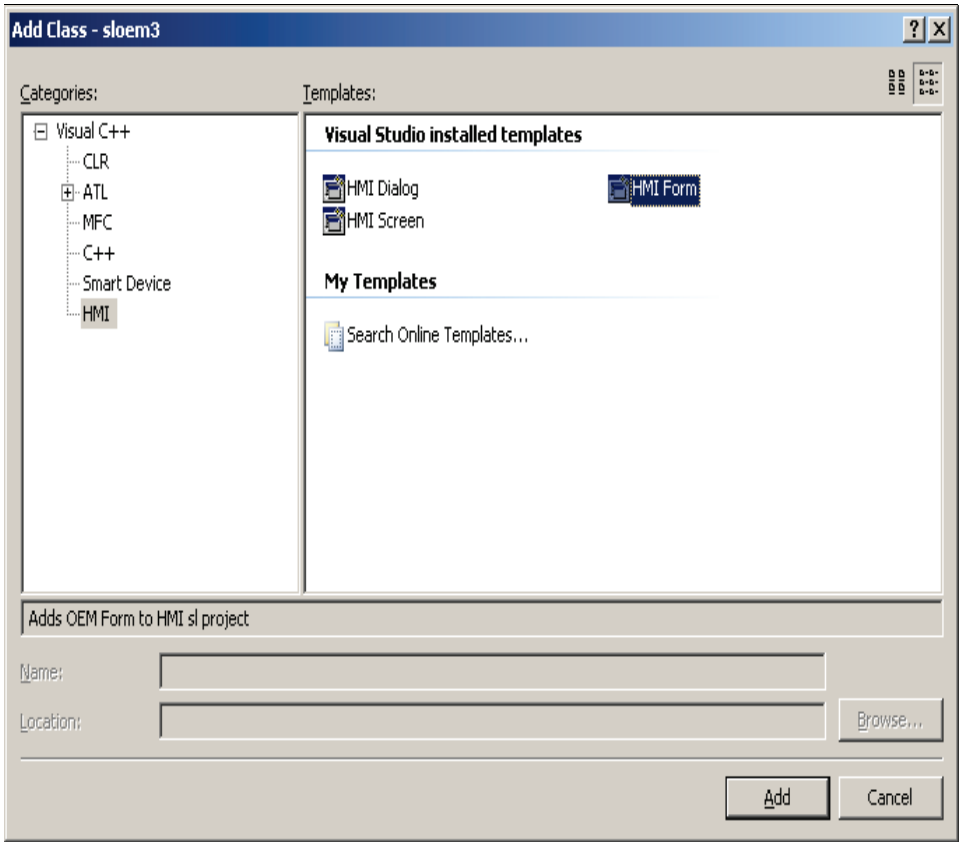


图2-7:插入一个对话框类、屏幕类或窗体类

在确认选择关闭该对话框后，您可以为该类及其CPP文件和头文件命名。在插入一个窗体类时，您还可以确定是否生成一份Ui文件，以便在Qt Designer中进行编辑。

窗体类对话框的外观如下：

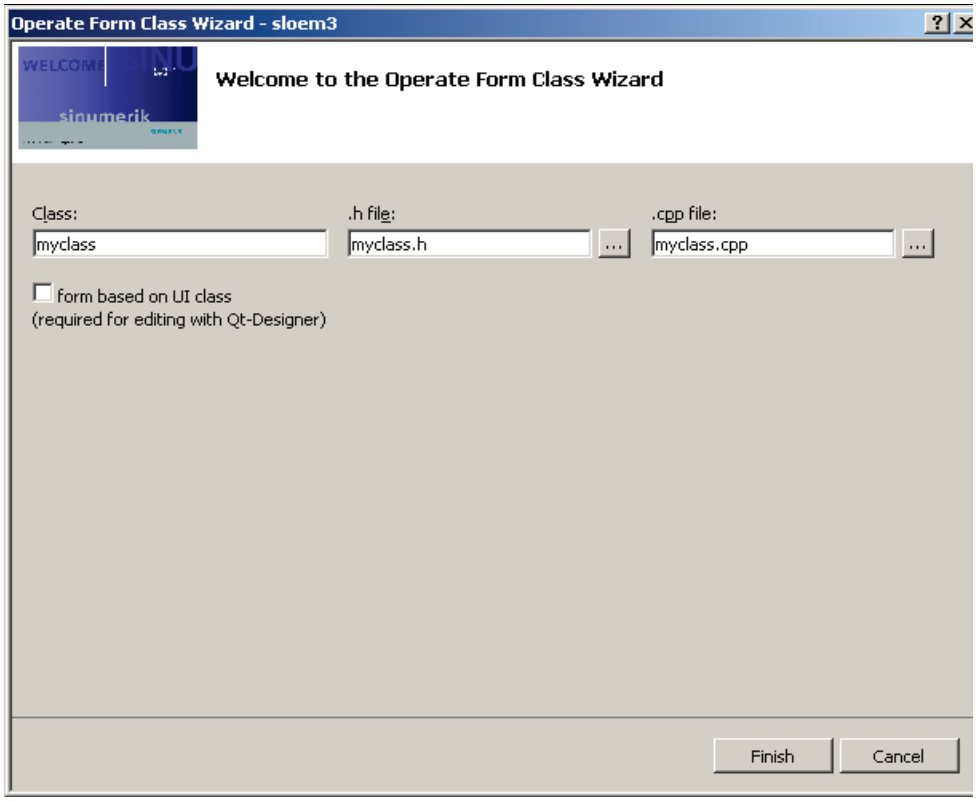


图2-8:插入一个窗体类

按下“Finish”按钮后，您选中的类成功命名并成功创建。现在您可以直接生成SINUMERIK Operate项目，因为所有必要的设置都会自动完成。

1) **类文件**（CPP文件和头文件）自动生成，添加到项目中。

2) **UI文件/MOC文件**同样自动生成，添加到项目中。

如果为某个窗体类选择了支持Qt Designer编辑，项目文件夹“Forms”下会自动生成一份Ui文件。用Qt Designer打开该文件后，您会看到一个空窗体，标签为“QLabel”。

为支持Qt Designer中的编辑，文件夹“Generated”中生成了对应的MOC文件和Ui头文件。但这些文件只能在创建了SINUMERIK Operate项目之后才存在，在此之前不能打开。

插件文件中为每个新建的类创建了一个导出宏命令。

3) 如果有一份**PRO文件**，此处会添加新文件（CPP文件/头文件/UI文件）。

插入自定义的Qt类（带Q_OBJECT宏命令）

需要在现有SINUMERIK Operate项目中插入一个自定义的的Qt类时，除了CPP文件和头文件外，您还需要一份MOC文件。 该文件可通过头文件的“Custom Build Step”设置自动生成。

合适的“Custom Build Step”参见章节
“AE-Service\SIExAeEventSource”的说明。
此处您可以复制文件“slexaeeventsourceform.h”中的“Custom Build Step”。

同理，在第一次生成的MOC文件也要添加到SINUMERIK Operate项目中。

插入自定义文件

需要在现有SINUMERIK Operate项目中插入更多自定义文件时，可能需要手动设置必要的“Custom Build Step”，比如：当您需要补充设置语言支持时。

此时您可以复制示例，采用其中的“Custom Build Step”。在SINUMERIK Operate编程包中可能会出现以下“Custom Build Step”：

表2-2：Custom Build Step（通用）

类型	示例	Custom Build Step
语言支持（TS文件）	GUIFrameWork\SIExGuiLanguage	slexguilanguage_deu.ts
对话框配置（XML文件）	GUIFrameWork\SIExGuiLanguage	slexguilanguage.xml
将系统配置文件复制到“Output”	GUIFrameWork\SIExGuiLanguage	systemconfiguration.ini
Moc’ing（带有Q_OBJECT宏命令的类）	GUIFrameWork\SIExGuiLanguage	slexguilanguageform.h
Uic’ing（UI窗体）	GUIFrameWork\DesignerSample1	designersample1_form1.ui
添加更多帮助手册	GUIFrameWork\SIExGuiOnlineHelp	slhlp.xml
添加更多屏幕布局	GUIFrameWork\SIExGuiSoftkeyBars	slexscreenlayout.xml

表2-3：Custom Build Step（仅限报警与事件）

类型	示例	Custom Build Step
报警源	AE服务\SIExAeEventSource	slexaeeventsource_db.xml
添加更多报警源	AE服务\SIExAeEventSource	slaesvcconf.xml
添加更多报警文本	AE服务\SIExAeEventSource	slaesvcadapconf.xml

2.2 执行项目

在Visual Studio中生成项目后，所有和执行应用程序相关的文件都位于目录debug\output或release\output下。子目录(appl, cfg, lng, hlp)使目标目录中文件的保存情况一目了然。

项目通常包含以下生成的文件：

- DLL文件，包含了对话框类、屏幕类和窗体类，比如：sloem1dialog.dll。
- 经过转换的XML对话框配置文件，转换后的扩展名为*.hmi，比如：sloem1dialog.hmi。
- 经过转换的语言文件，转换后的扩展名为*.qm，比如：sloem1dialog_deu.qm。

您可以激活事先准备好的“Post-Build Event”，以便直接将生成的文件复制到您PC的OEM目录 (\hmisl\oem\sinumerik\hmi\)下：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties/Build Events/Post-Build Event”
4. 将“Excluded From Build”设为“No”

执行项目要求您将自定义对话框集成到SINUMERIK Operate软件中。为此您必须将Visual Studio向导生成的systemconfiguration.ini文件复制到目录\hmisl\oem\sinumerik\hmi\cfg中。如果Visual Studio向导没有生成该文件，您必须自行生成该文件。相关信息和示例参见章节4.17。

点击以下菜单项，启动SINUMERIK Operate：

开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → HMI

Visual Studio的配置

在开发PC上：
➔ 在该设备上可以运行调试版和释放版项目。项目调试的相关信息见章节2.3。

在PCU50上：
➔在该设备上只能运行经过转换的释放版Visual Studio项目。

分辨率

需要在开发PC上用各种分辨率来测试项目时，您可以使用工具“slrsresolutionswitch”。利用该工具您可以在SINUMERIK Operate运行时动态切换分辨率。该工具的设置菜单项为：

开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → 工具 → Change Resolution→640 (800 / 1024 / 1280)

2.3 调试项目

修改系统配置文件

项目生成时会生成一份DLL文件，它包含了对话框类、屏幕类和窗体类。不允许直接执行和调试该DLL文件，在任何情况下都要首先通过HMI Host进程加载该文件。

因此在利用Visual Studio调试项目时，必须首先启动HMI Host进程，该进程随后载入自定义DLL文件。HMI Host进程在标准HMI的systemconfiguration.ini文件中定义。为避免该进程随系统直接启动，您必须修改systemconfiguration.ini，使进程稍后由Visual Studio启动。

用文本编辑器打开文件“\hmisl\oem\sinumerik\hmi\cfg\systemconfiguration.ini”，插入以下数据行：

```
[processes]
PROC001= image:=slsmhmihost, process:=SlHmiHost1, cmdline:="-
ORBCollocationStrategy direct", deferred:=true
```

deferred:=true时，HMI Host 进程不会随系统直接启动。

Visual Studio中的调试设置

设置目标是使HMI Host进程由Visual Studio启动。为此打开Visual Studio项目的属性窗口。 选择左侧树形图中的条目“Debugging”，在“Command”中输入以下数值：

```
$(HMI_SL_PP_SUBST)\hmisl\siemens\sinumerik\hmi\base\slsmhmihost.exe
```

在条目“Working Directory”中输入以下数值：

```
$(HMI_SL_PP_SUBST)\hmisl\siemens\sinumerik\hmi\base
```

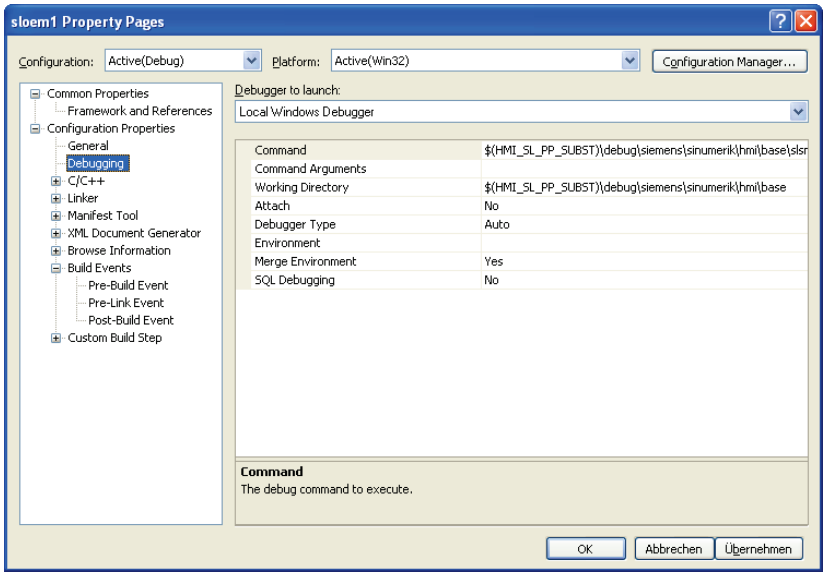


图2-9:项目调试设置

开始调试

首先启动HMI，以开始调试。为避免调试期间出现超时错误，必须利用一个特殊选件启动HMI。

该选件是：

slsmsystemmanager.exe --no-timeout

选择开始菜单中SINUMERIK Operate编程包中的条目HMI (no Timeout)，即可选中该选件。

现在HMI成功启动后启动HMI Host进程，随后显示：“Startup area will be deferred loaded!”

现在您可以按下F5或通过菜单项
“Debug / Start Debugging”启动Visual Studio项目。

注

在Visual Studio项目中应使用配置“Debug”。

SIQCap service中advise slot的特殊性

在调试SIQCap service中的advise slot时，可能会出现连接中断错误“8030009B - COS/CP interrupted connection by lifesign”。这通常是因为您调试advise slot的时间过长，超出20秒。在退出advise slot后，连接会自行恢复，但您会收到一个连接中断信号和连接恢复信号。

这两个信号的输出可以被封锁。为此打开文件：

```
$(HMI_SL_PP_SUBST)\hmisl\siemens\sinumerik\hmi\base\cp_param.ini
```

在其中添加下述条目：

```
[L4_Srv0]
Timeout = 0

[L4_Srv1]
Timeout = 0
```

这一步骤仅限开发PC。

图2-10:开始调试时的HMI闪屏

现在在Visual Studio中通过菜单项
Debug → Start Debugging (F5)开始调试。
HMI Host进程启动，载入所有服务和对话框。现在您可以在自己的源代码中设置暂停点，使程序在特定位置上暂停。

结束调试

您可以通过操作区域菜单或者关闭系统管理器窗口来关闭HMI，结束调试。检查在Visual Studio中是否已结束了调试，如果还没有结束，选择菜单项→ Stop.。

接着检查HMI中的所有进程是否已结束。为此启动Windows任务管理器。
不允许再运行以下进程：slsmsystemmanager.exe, slsmhmihost.exe 和 cp_840di.exe。

2.4 将应用程序传送到目标系统PCU50中

在成功生成应用程序后，将组件复制到目标系统PCU50中。在这一步骤中要传送下述文件：

注

在传送文件前，您首先要转换释放版的Visual Studio项目。 所有和执行应用程序相关的文件都位于Visual Studio项目的目录“release\output”下。

F:/hmisl/oem/sinumerik/hmi/appl

该目录保存了库文件（DLL文件）、经过转换的配置文件（HMI文件）和其他所有无法归类到其他目录中的文件。

F:/hmisl/oem/sinumerik/hmi/cfg

该目录保存了通过SlHmiSettingsQt类的设置机制（详见章节4.16）读写的配置文件。因此其中也保存了systemconfiguration.ini文件。传送自定义的systemconfiguration.ini文件时要注意，不要覆写可能有的其他OEM应用程序的此类文件。

F:/hmisl/oem/sinumerik/hmi/hlp

该目录保存了在线帮助的相关数据。

F:/hmisl/oem/sinumerik/hmi/ico/ico640

该目录保存了分辨率为640*480的图片。

F:/hmisl/oem/sinumerik/hmi/ico/ico800

该目录保存了分辨率为800*600的图片。

F:/hmisl/oem/sinumerik/hmi/ico/ico1024

该目录保存了分辨率为1024*768的图片。

F:/hmisl/oem/sinumerik/hmi/ico/ico1280

该目录保存了分辨率为1280*960的图片。

F:/hmisl/oem/sinumerik/hmi/ico/ico1600

该目录保存了分辨率为1600*1200的图片。

F:/hmisl/oem/sinumerik/hmi/lng

该目录保存了和语言相关的文本文件（QM文件）。

注

此外，OEM应用程序有可能保存在自定义的OEM子目录下。 详细说明参见章节4.20“OEM子目录和版本”

2.5 通过公司网络(-X130)和外部HMI通讯

2.5.1 概述

SINUMERIK Operate编程包中提供了相关功能，可以通过接口–X130（公司网络）和840D sl控制系统连接在一起。

因此您可以将您的开发PC和840D sl控制系统集成到公司网络中，
以便从开发PC出发访问所有HMI服务或OA接口。

为符合信息安全要求，840D sl的NCU有内置防火墙。
您要首先打开相应的通讯端口，才能通过–X130实现通讯。

2.5.2前提条件

连接建立要满足以下前提条件：

1. NCU Base (LinuxBase)的版本起码是V02.20.03.00。您可以点击“诊断”操作区中的“版本”软键，查看版本信息。

2. 开发PC和840D sl的 –X130端口必须位于同一个网络中（子网）。

2.5.3 工作步骤 1

按照下述步骤建立连接：

1. “mmc.ini”文件中[NCU840D]段中的IP地址必须和–X130的地址一致，即公司网络中NCU的IP地址或计算机名称。

2. 外部开发PC可通过PUTTY登录嵌入式Linux系统：
   用户➔ ’manufact’
   密码➔ ’SUNRISE’（或者当前制造商密码）

3. 写入以下命令，打开开发PC和840D sl控制系统之间的通讯端口：

   sc openport HMIEXT

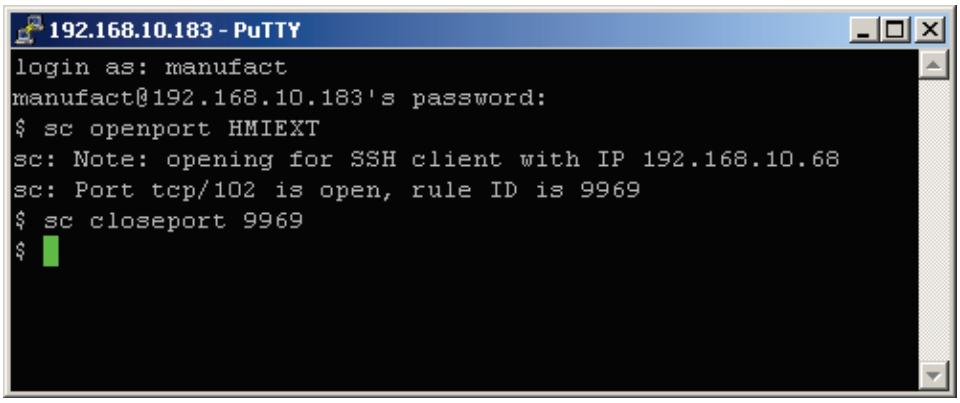
   利用参数“-timeout”您可以设置端口打开的时间，单位为分钟。
   没有设置时间时，缺省打开时间为10分钟。最长可设为60分钟。例如要设置35分钟的打开时间：

   sc openport -35 HMIEXT
4. 您也可以在该时间期满前关闭端口。为此写入命令：

   sc closeport ID

   ID是一个由“sc openport HMIEXT”返回的四位数编号。

例如：写入的命令可以为：



注

命令“sc closeport ID”不会断开开发PC和840D sl之间现有的连接，它只是不允许新建连接。重启开发PC上的HMI后，需要再次写入命令“sc openport HMIEXT”。

注释：
您也可以通过程序中的SSH命令打开端口。

2.5.4 工作步骤 2

按照下述步骤建立永久连接：

1. “mmc.ini”文件中[NCU840D]段中的IP地址必须和–X130的地址一致，
   即公司网络中NCU的IP地址或计算机名称。

2. 外部开发PC可通过诸如WinSCP之类的SCP客户端程序登录嵌入式Linux系统：
   用户➔ ’manufact’
   密码➔ ’SUNRISE’ （或者当前制造商密码）

3. 现在您可以在文件“/card/user/system/etc/basesys.ini”中修改下述条目，打开端口5900、5901和102：

```
[LinuxBase]
FirewallOpenPorts="TCP/5900 TCP/5901 TCP/102"
```

4. 重启嵌入式Linux系统后，修改才生效。

3 SINUMERIK Operate Linux

3

本章主要内容

本章旨在向您介绍如何生成嵌入式-- Linux系统适用的应用程序以及如何将程序传送到系统中。

3.1 生成嵌入式Linux系统适用的应用程序

将需要生成的嵌入式Linux系统适用的源代码文件保存到目录“\work\workspace”下。关联的qmake项目文件(*.pro)也必须保存到该目录下。项目文件的句法详见对应的QT文档。在“CAP-Service\SIExCapSyncMap”示例中句法应为：

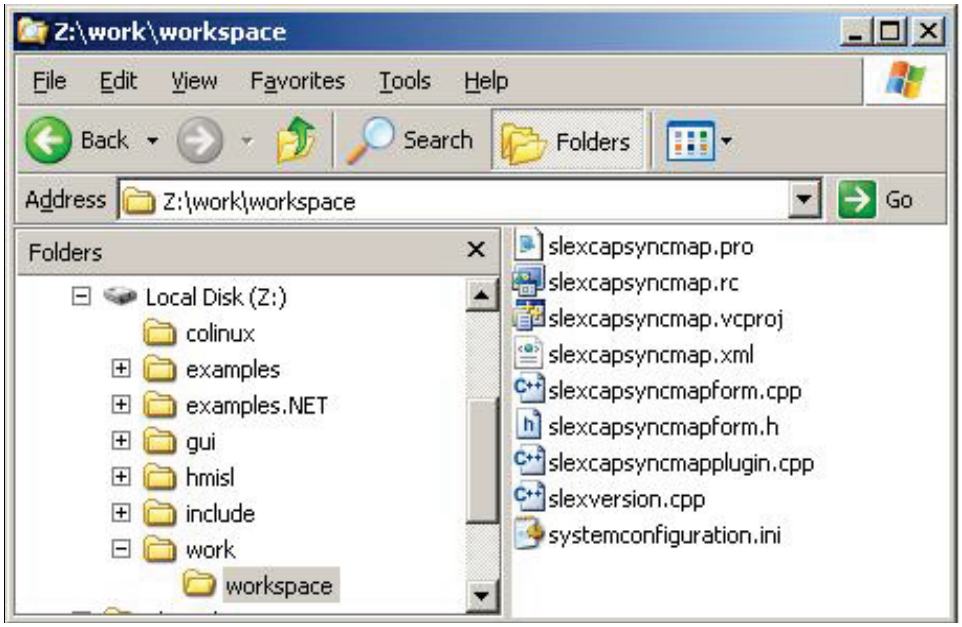


图3-1:生成应用程序前的SIExCapSyncMap

接着进入下述开始菜单项：

开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → build Linux

在生成应用程序后，SIPPLogViewer会显示一段总结，结果保存在“\work\from_linux_release\”下。SIPPLogViewer的三个区域从上至下显示以下内容：

a) VCProj文件和PRO文件之间的对比
   ➔ 指出导入的库、头文件和CPP文件之间的不同之处。

b) 文件“qmake.log”
   ➔
   包含了“qmake”命令的结果，该命令从一份PRO文件生成一份所谓的Makefile。点击“…open file”，可打开qmake.log的详细说明。

c) 文件“make.log”
   ➔包含了“make”命令的结果，该命令对源代码文件进行编译、链接，最后生成目标文件（SO文件）。点击“…open file”，可打开make.log的详细说明。

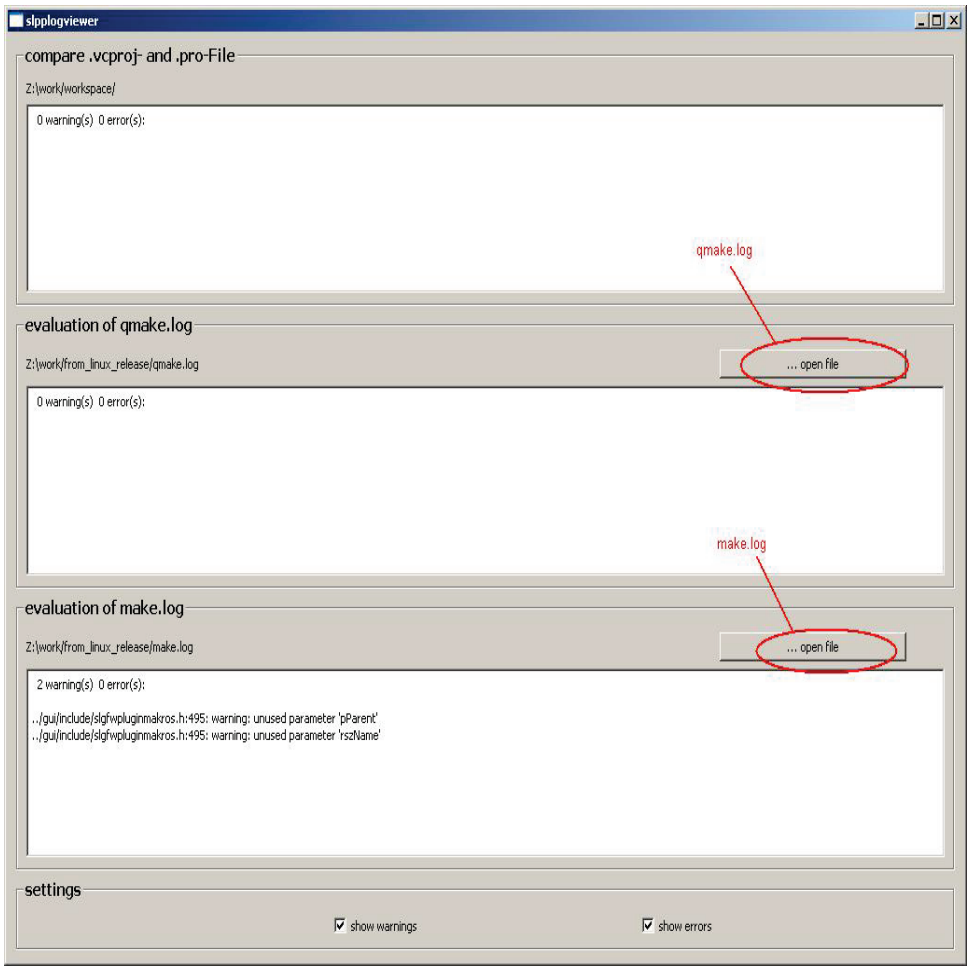


图3-2:slPPLogViewer（生成应用程序后的的SlExCapSyncMap）

如果“qmake.log”和“make.log”都没有报错，而且有SO文件，则表明应用程序成功生成。接下来便可以将所需文件通过诸如WinSCP之类的WindowsSCP客户端程序传送到嵌入式Linux中了。

生成的SO文件位于目录“\work\from_linux_release\release\output\appl\”下。

3.2 将应用程序传送到嵌入式Linux系统中

在成功生成应用程序后，便可以利用一个SCP客户端程序（比如WinSCP）将应用程序传送到嵌入式Linux系统中。

登录数据：
用户➔ 'manufact'
密码➔ 'SUNRISE' （或者当前制造商密码）

在这一步骤中要传送下述文件：

注

在传送文件前，您首先要转换释放版的Visual Studio项目。 所有和执行应用程序相关的文件都位于Visual Studio项目的目录"release\output"下，除了SO文件，该文件的生成方式见章节3.1。

/card/oem/sinumerik/hmi/appl

该目录保存了库文件（SO文件）、经过转换的配置文件（HMI文件）和其他所有无法归类到其他目录中的文件。

/card/oem/sinumerik/hmi/cfg

该目录保存了通过SlHmiSettingsQt类的设置机制（详见章节4.16）读写的配置文件。因此其中也保存了systemconfiguration.ini文件。传送自定义的systemconfiguration.ini文件时要注意，不要覆写可能有的其他OEM应用程序的此类文件。

/card/oem/sinumerik/hmi/hlp

该目录保存了在线帮助的相关数据。

/card/oem/sinumerik/hmi/ico/ico640

该目录保存了分辨率为640*480的图片。

/card/oem/sinumerik/hmi/ico/ico800

该目录保存了分辨率为800*600的图片。

/card/oem/sinumerik/hmi/ico/ico1024

该目录保存了分辨率为1024*768的图片。

/card/oem/sinumerik/hmi/ico/ico1280

该目录保存了分辨率为1280*960的图片。

/card/oem/sinumerik/hmi/ico/ico1600

该目录保存了分辨率为1600*1200的图片。

/card/oem/sinumerik/hmi/lng

该目录保存了和语言相关的文本文件（QM文件）。

注

此外，OEM应用程序有可能保存在自定义的OEM子目录下。 详细说明参见章节4.20“OEM子目录和版本”

3.3 故障排查

如果程序在控制系统上导致异常，您可以采取以下几种方法来排查故障，下文会详细介绍：

斜升

如果在HMI启动阶段就出现故障，通常您只需查看HMI的日志文件，即可了解故障详情该文件位于控制系统的目录/card/user/sinumerik/hmi/log/hmi下。系统一旦启动，文件slsmsystemmanager.log便开始记录。启动时出现的问题便会记录到该文件中。您可以借助工具“errorlookup”来了解文件中故障号的具体含义。
可在开始菜单项中找到该工具：

开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → 工具→ errorLookup

日志文件

如果HMI正在运行时突然关机，控制系统的目录/card/user/sinumerik/hmi/log/hmi下会备有一份关机故障日志，格式为“slsmhmihost_crash_<date>T<time>.log”。通常情况下该日志足以帮助您确定出现故障的位置。

跟踪

另外，您还可以将“TRACE输出”集成到程序中。如何集成以及如何启动此类TRACE详见章节4.21“TRACE输出”。

3.4 后台处理信息

如需获得更多关于coLinux的信息请访问网址
http://wiki.colinux.org/wiki/Main_Page

批处理文件“make.bat”首先在“\work”下生成一份“buildmode.txt”文件，其中包含了所需的buildmode。接着coLinux启动。在Linux文件系统下的“/mnt/windows”中创建“\work”目录。在Linux系统启动后，复制“\work”下窗口侧的文件“config.txt”，并接着执行该文件。在当前提供的该文件版本中，首先设置buildmode，接着将“\work\workspace”的所有内容复制到Linux中，生成一份Makefile用于执行make进程，接着复制所有文件到“\work\from_linux_<BUILDMODE>”中。之后Linux系统再次关闭。最后界面从待调用的批处理文件中关闭。

为了直接在Linux系统中开展工作，您可以删除文件“config.txt”中命令“return 0”前的注释符：#。此时文件执行会在该位置上终止，提供一个Linux外壳程序供使用。您可以输入“halt”关闭系统。

注

SINUMERIK Operate编程包目前只能借助coLinux生成释放版的应用程序。

4

4 GUI-组件

本章主要内容

本章主要向您介绍 SINUMERIK Operate 的 GUI 开发理念、如何利用 GUI Framework 开发出自己的 GUI 组件以及如何将该组件集成到现有 HMI 中。

本章也会涉及到语言文本，介绍如何读配置数据、如何访问文件以及如何生成跟踪文件来进行故障排查。

4.1 GUI 开发理念

操作界面首先由若干个操作区域组成。这些操作区域通常是按照其功能划分的。

操作区域又由对话框组成。在最简单的设计中，一个操作区域只有一个对话框。而功能丰富的操作区域可以由多个对话框组成，这些对话框之间通过“浏览”的方式相互嵌套。

对话框又可以继续细分为多个操作画面，即所谓的“屏幕”。屏幕再分为一个或多个用于输入数据的窗体、一条消息栏、多个水平软键和多个垂直软键。软键可用于在一个对话框的多个屏幕之间来回切换、显示/隐藏窗体和选择某项功能。

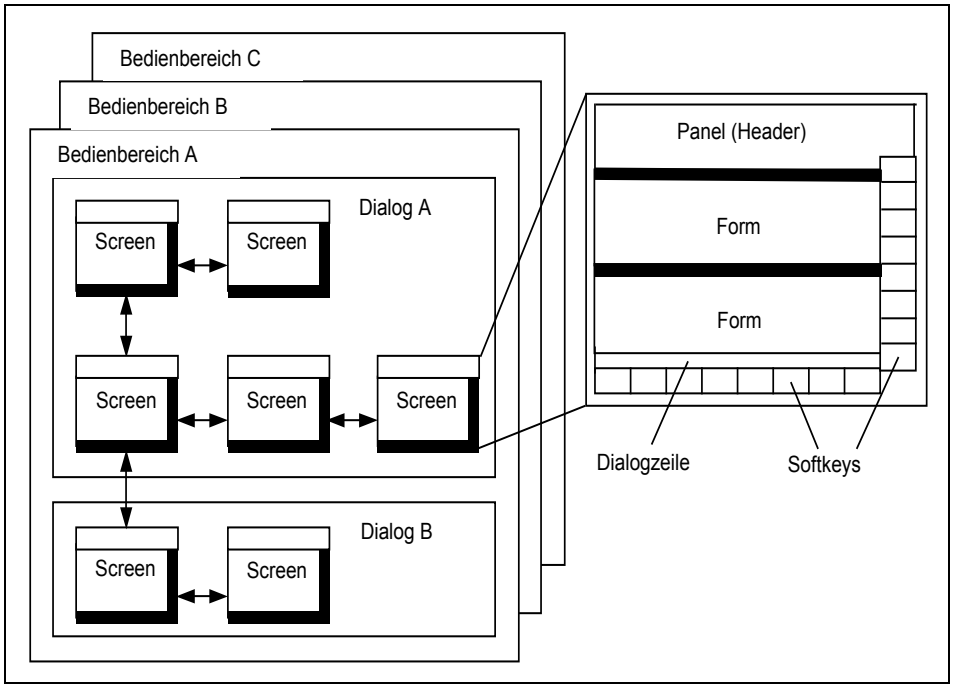


图 4-1:HMI GUI 开发理念

4.2 GUI Framework

图形用户界面的开发基本上分为两个阶段：定义阶段和实现阶段。

组态

您可以自行定义 HMI 对话框的用户界面外观。在定义阶段您需要定义一个 HMI 对话框要包含哪些屏幕、一个屏幕要包含那些软键和窗体以及按下软键后要执行哪个任务。

一个对话框只需要定义一次，该定义会针对所有下属屏幕生效。

GUI 定义在一份 XML 文件中进行，该文件必须转换为二进制格式，以适应 runtime 环境。

实现

在编码阶段，您需要利用 C++ 编程语言在 GUI 组件中实现切换屏幕、显示/隐藏窗体产生的功能。

您可以利用 GUI Framework 来开发 GUI 组件：对话框、屏幕和窗体。GUI Framework 包含了这些组件的基本类，基本类定义了需要在 GUI 组件中实现的接口。

通过实现或改写这些接口，组件可以对不同的事件作出响应。

如果对于某个 GUI 组件的某个实例而言不需要实现，您也可以在程序运行的同时直接使用该组件的基本类。

GUI 组件的实现是在库（DLL 或共享库）中进行的。您可以在多个库中或者统一在一个库中实现。通常在一个库中即可实现一个对话框及其下属屏幕和窗体。
在程序运行时，GUI 组件是通过每个库中的插件机制实例化的。参见图 4-2。

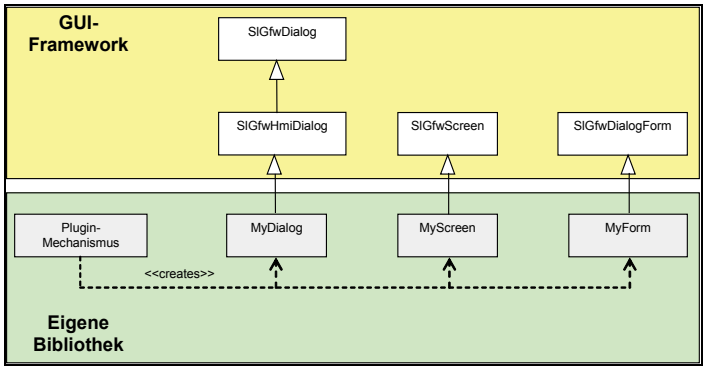


图 4-2:GUI Framework

插件机制

在系统运行时，诸如 HMI 对话框之类的 GUI 组件是利用插件机制集成到 SINUMERIK Operate 的 HMI 系统中的。您必须为每个包含 GUI 组件的 DLL/库实现插件机制。
插件机制在一个单独的源代码文件（比如：plugin.cpp）中实现。GUI Framework 为插件机制的实现提供以下宏命令：

表 4-1：插件宏命令

宏	描述
SL_GFW_BEGIN_PLUGIN_EXPORT	开始插件机制的实现。
SL_GFW_END_PLUGIN_EXPORT	结束插件机制的实现。
SL_GFW_DIALOG_PLUGIN_EXPORT	导出一个对话框类。该对话框类的名称传送给宏命令。
SL_GFW_SCREEN_PLUGIN_EXPORT	导出一个屏幕类。该屏幕类的名称传送给宏命令。
SL_GFW_DIALOGFORM_PLUGIN_EXPORT	导出一个窗体类。该窗体类的名称传送给宏命令。

在使用这些宏命令时必须写入头文件 slgfwpluginmakros.h。

例：

```
#include "slgfwpluginmakros.h"
#include "slexdialog.h"
#include "slexscreen.h"
#include "slexform.h"

SL_GFW_BEGIN_PLUGIN_EXPORT()
    SL_GFW_DIALOG_PLUGIN_EXPORT(SlExDialog)
    SL_GFW_DIALOGFORM_PLUGIN_EXPORT(SlExForm)
    SL_GFW_SCREEN_PLUGIN_EXPORT(SlExScreen)
SL_GFW_END_PLUGIN_EXPORT()
```

在本例中，宏命令导出了对话框类 SlExDialog、窗体类 SlExForm 和屏幕类 SlExScreen 。GUI Framework 的插件机制可以在程序运行时创建这些类的对象。

检查库

在使用 GUI Framework 时，要检查项目设置，确保库 `slgfw.lib`，`slgfwwidget.lib`，`slhmiutilitieslib.lib` 和 `slgrgrid.lib` 添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”。

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
    -lslgfw -lslgfwwidget -lslhmiutilitieslib -lslgrgrid
```

4.3 屏幕布局

HMI 对话框的外观涉及到屏幕布局“screenlayout”这一概念。屏幕布局决定了以何种大小显示窗体和软键。您可以为一个对话框的每个下属屏幕设计出单独的布局。

通过调整屏幕布局您可以调整操作界面的外观，使它和不同操作面板上数量不同的软键条相协调。另外，它还可以用于调整不同操作区内标题栏的大小。

GUI Framework 为您提供一份标准的屏幕布局，它是针对有两条软键条的操作面板开发的。您可以据此开发出自己的屏幕布局，设计出独特的窗体大小。

标准屏幕布局的保存目录为：

```
[Install-
Dir]\hmisl\siemens\sinumerik\hmi\base\slstandardscreenlayout.xml(hmi)
```

…和…

```
[Install-Dir]\hmisl\siemens\sinumerik\hmi\base\sl2skbscreenlayout.xml(hmi)
```

其中“sl2skbscreenlayout”包含了软键（位置、大小和硬键）、对话条和 ETC 按键的定义。“slstandardscreenlayout”除此以外还包含了几个预定义的窗体面板（Form panel）。

屏幕布局的组成部分

一个屏幕布局由多个部分组成：

窗体区域（Form area）
本部分定义在哪个区域中显示屏幕的窗体。

窗体面板（Form panel）
本部分定义屏幕下属窗体的位置和大小。它简化了屏幕或窗体的设计，因为在设计屏幕时只需要确定窗体所在的“面板”，而不需要确定窗体坐标和大小 (*x*, *y*, *width*, *height*) 。

“SlStandardScreenLayout”已经包含几个预定义的窗体面板，您可以直接使用这些面板，无需开发出自己的屏幕布局：

已批准
Sheep , 19:32:53, 2015/07/21

- “FullForm”和“FullFormAdaptable”
- “LeftForm”和“RightForm”
- “ModalForm”、“SmallModalForm”、“ModalFindForm”、“ModalReplaceForm”、“ModalNumberForm”和“ModalSettingForm”
- “UpperForm”和“LowerForm”

不同分辨率的屏幕上窗体的位置和大小可参见文件“slstandardscreenlayout.xml ”。

软键条（Softkey bar）
软键条是由多个左右排列或上下排列的软键（按钮）构成的。

软键
本部分定义了软键条内软键的位置和大小。

对话条（Dialog bar）
本部分定义对话条的位置和大小。

图 4-3 展示了屏幕布局的组成部分以及各部分的相互关系。一个屏幕布局通常由一个窗体区域（含多个窗体面板）、多个软键条（含多个软键）和一个对话条组成。

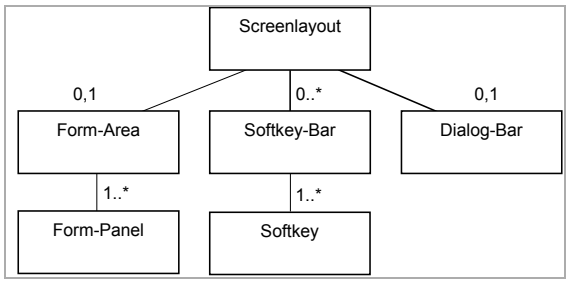


图 4-3:屏幕布局的组成部分

屏幕布局的定义

屏幕布局在一份 XML 文档中定义。该 XML 文档的结构和图 4-3 展示的结构一致：

```
<SCREENLAYOUTS>
  <SCREENLAYOUT id, resolution, extends>
    <FORMAREA x, y, width, height>
      <FORMPANEL id, x, y, width, height />
      ... 更多窗体面板的定义
    </FORMAREA>
    <SOFTKEYBAR id, x, y, width, height, orientation>
      <SOFTKEY hardkey, position, x, y, width, height />
      ... 更多软键的定义
    </SOFTKEYBAR>
    ... 更多软键条的定义
    <DIALOGBAR x, y, width, height />
    <ETCKEY softkeybar />
  </SCREENLAYOUT>
  ... 更多屏幕布局的定义
</SCREENLAYOUTS>
```

该 XML 文件可以包含多个屏幕布局的定义。屏幕布局通过其 **ID** 和**屏幕分辨率**加以区分。因此，一份 XML 文件可以包含所有分辨率条件下一个对话框的屏幕布局定义。

如果没有找到当前分辨率适合的屏幕布局，GUI Framework 会使用具有相同 ID 的屏幕布局，据此换算出当前布局各个组成部分的位置和大小。

设计屏幕布局

您可以在现有屏幕布局基础上设计出自己的屏幕布局。

通常对话框开发人员设计出自己的屏幕布局，以定义新的窗体面板（窗体的位置和大小）。

GUI Framework 为此提供了一份标准屏幕布局文件“Sl2SKBScreenLayout”，但它只定义了所有分辨率条件下软键和对话条的位置和大小。您可以在该标准布局基础上增加窗体区域和窗体面板。

使用<SCREENLAYOUT>标签中的属性 **extends** 进行设计。该属性确定了要设计哪个屏幕布局。句法为**文件名.LayoutID**。 完成该设计后，系统运行时会首先加载标准屏幕布局，然后加载经过设计的屏幕布局，将这两个布局整合在一起。

如果您设计的屏幕布局有些部分的 ID 和标准布局的一样，系统会采用您设计的屏幕布局。

示例：

```
<SCREENLAYOUTS>
  <SCREENLAYOUT id="MyLayout" resolution="640x480"
         extends="sl2skbscreenlayout.Sl2SKBScreenLayout">
    <FORMAREA x="0" y="33" width="559" height="394">
      <FORMPANEL id="MyForm" x="0" y="0" width="559" height="394"/>
    </FORMAREA>
  </SCREENLAYOUT>
  <SCREENLAYOUT id="SlStandardScreenLayout" resolution="800x600"
         extends="sl2skbscreenlayout.Sl2SKBScreenLayout">
    <FORMAREA x="0" y="41" width="699" height="493">
      <FORMPANEL id="MyForm" x="0" y="0" width="699" height="493"/>
    </FORMAREA>
  </SCREENLAYOUT>
  <SCREENLAYOUT id="SlStandardScreenLayout" resolution="1024x768"
         extends="sl2skbscreenlayout.Sl2SKBScreenLayout">
    <FORMAREA x="0" y="52" width="895" height="630">
      <FORMPANEL id="MyForm" x="0" y="0" width="895" height="630"/>
    </FORMAREA>
  </SCREENLAYOUT>
</SCREENLAYOUTS>
```

在本例中，设计是在文件“sl2skbscreenlayout”中的标准屏幕布局基础上进行的。该布局只包含两个软键条和一个对话条。 经过设计后，所有分辨率条件下的标准布局都增加了一个窗体面板“MyForm”。

如果您还需要使用“SlStandardScreenLayout”中预定义的窗体面板（比如：“FullForm”或“LeftForm”等面板），不要写入：

```
extends="sl2skbscreenlayout.Sl2SKBScreenLayout"
```

...而应写入：

```
extends="SlStandardScreenLayout.SlStandardScreenLayout"
```

在“GUIFrameWork\SIExGuiSoftkeyBars”的示例目录下有一个示例，指出如何处理第二个水平或垂直软键条。

4.4 HMI 对话框

术语解释

每个操作区域都至少由一个对话框组成，对话框又由多个操作画面（屏幕）组成。对话框是一个纯逻辑组件。

对话框读取对话框配置文件，生成相应的屏幕、窗体和软键。 它会对发生的事件作出响应，如：软键被按下或语言被切换等事件。

用户界面的结构

图 4-4 展示了一个对象模型，它是定义 HMI 对话框用户界面的基础。一个对话框用户界面由一个或多个屏幕组成。而屏幕又通常由一个或多个窗体、一个或多个菜单（软键条）和一个对话条组成。菜单由一个或多个菜单级组成，每个菜单级由一个或多个软键组成。每个菜单级可定义的最大软键数量是由与该菜单级相关联的软键条上软键的数量决定的。

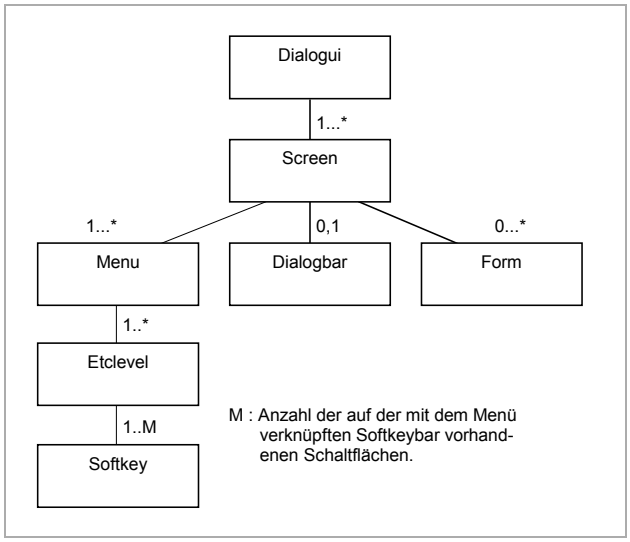


图 4-4:用户界面的结构

用户界面的定义

用户界面在一份 XML 文档中定义。该 XML 文档的结构和图 4-4 展示的对象模型一致：

```
<DIALOGUI 对话框 GUI 属性>
  <SCREEN 屏幕属性>
    <PROPERTY name="propertyname">value</PROPERTY>
    ... 更多屏幕属性的定义
    <FORM 窗体属性/>
      <PROPERTY name="propertyname">value</PROPERTY>
      ... 更多窗体属性的定义
      </FORM>
    ... 更多窗体的定义
    <MENU 菜单属性>
      <ETCLEVEL Etclevel 属性>
        <SOFTKEY 软键属性>
          <PROPERTY name="propertyname">value</PROPERTY>
          ... 更多软键属性的定义
          <NAVIGATION 浏览属性>
          ... 浏览标签
          </NAVIGATION>
          <FUNCTION 功能属性 >
          ... 功能标签
          </FUNCTION>
          ... 更多功能的定义
        </SOFTKEY>
        ... 更多软键的定义
      </ETCLEVEL>
      ... 更多 Etclevel 的定义
    </MENU>
    ... 更多菜单的定义
    <RECALL 回调属性>
      <NAVIGATION 浏览属性>
      ... 浏览标签
      </NAVIGATION>
    </RECALL>
  </SCREEN>
  ... 更多屏幕的定义
  <DIALOGBAR 对话条属性/>
</DIALOGUI>
```

XML 标签 DIALOGUI

XML 标签 DIALOGUI 引入 HMI 对话框用户界面的定义。XML 标签 DIALOGUI 支持以下属性：

表 4-2：XML 标签 DIALOGUI

属性	描述
screenlayout	指定用哪个屏幕布局显示对话框。定义屏幕时确定的屏幕布局会改写此处指定的布局。
panellayout	指定用哪个面板布局显示对话框的下属面板。
textfile	指定包含对话框语言文本的文件名称。
textcontext	指定 *textfile* 属性指出的文件中对话框语言文本的命名空间（context）。*textfile* 和 *textcontext* 这两个属性总是要成对指定。
defaultscreen	指定在首次选择某个对话框但没有选择某个屏幕时系统自动显示的屏幕。

属性	描述
helpdocument	指定本对话框的在线帮助文件。当不管是具有输入焦点的窗体还是该窗体所属的屏幕都没有定义在线帮助时，按下“INFO”键后显示该文件的内容。
helpanchor	指定从哪个 HTML 锚点开始显示 *helpdocument* 属性指出的在线帮助文件。

对话框类

您可以通过实现一个对话框类在一个 HMI 对话框中插入自定义的功能。

每个 HMI 对话框类必须由 GUI Framework 的 SlGfwHmiDialog 类导出。SlGfwHmiDialog 不仅提供导出的对话框类的接口定义，而且提供基本实现。

通过改写 SlGfwHmiDialog 的虚拟方法，对话框可对特定事件作出响应。可用的方法参见本章末尾 SlGfwHmiDialog 的接口引用。

! **重要提示**

在改写一个虚拟方法时，必须一同调用基本类的实现。

以下情形下最好实现一个对话框类：

- 需要跨屏幕管理数据时。
- 需要处理屏幕通用的事件时，比如：onShowMenu() 用作对话框通用的菜单。
- 需要修改显示和隐藏对话框时的属性时。

示例：

```
#ifndef MY_DIALOG_H_INCLUDED
#define MY_DIALOG_H_INCLUDED

#include "slgfwhmidialog.h"

class MyDialog :public SlGfwHmiDialog
{
    Q_OBJECT;

public:
    MyDialog(QObject* pParent = 0,
             const QString& rszName = QString::null);

    virtual ~MyDialog(void);

    virtual int init(const QString& rszArgs = QString::null);

    virtual void onFunction(const QString& rszFunction,
                            const QString& rszArgs,
                            bool& rbHandled);
};

#endif // MY_DIALOG_H_INCLUDED
```

在上例中可以看到对话框类的头文件的内容。init()和 onFunction()这两个虚拟方法被改写。

插件宏命令

利用下述宏命令可以将对话框类通知给库中的插件机制：

SL_GFW_DIALOG_PLUGIN_EXPORT(<类名称>)

对话框类的名称作为参数指定。

示例(plugin.cpp)：

```
#include "slgfwpluginmakros.h"
#include "mydialog.h"

SL_GFW_BEGIN_PLUGIN_EXPORT()
    SL_GFW_DIALOG_PLUGIN_EXPORT(MyDialog)
    // …更多导出的类
SL_GFW_END_PLUGIN_EXPORT()
```

4.5 HMI 屏幕

术语解释

一个对话框包含了一个或多个操作画面，即所谓的“屏幕”（参见图 4-4）。每个屏幕又包含了一个或多个窗体以及菜单（软键）。屏幕是一个纯逻辑组件。

XML 标签 SCREEN

在对话框配置文件中，屏幕是用 XML 标签 SCREEN 来定义的。

XML 标签 SCREEN 支持以下属性：

表 4-3：XML 标签 SCREEN

属性	描述
implementation	屏幕的实现。实现的说明格式为： *library.class* *library*: 包含屏幕实现的库。 *class*: SIGfwScreen 导出的类的名称。 未指定该属性时，程序运行期间会实例化基本类 SIGfwScreen。 选择性指定
name	屏幕实例的名称
screenlayout	指定用哪个屏幕布局显示屏幕。该指定会改写在对话框定义中指定的屏幕布局。 选择性指定
textfile	指定包含屏幕语言文本的文件名称。 选择性指定
textcontext	指定 *textfile* 属性指出的文件中对话框语言文本的命名空间（context）。该指定会改写在对话框定义中指定的命名空间。 选择性指定
helpdocument	指定本屏幕的在线帮助文件。当具有输入焦点的窗体没有定义在线帮助时，按下“INFO”键后显示该文件的内容。 选择性指定
helpanchor	指定从哪个 HTML 锚点开始显示 *helpdocument* 属性指出的在线帮助文件。 选择性指定
execmode	屏幕的执行模式（可选设定）： *modeless*: 指定屏幕以非模态方式执行（缺省）。 *modal*: 指定屏幕以模态方式执行。之前的屏幕仍旧可见，但是不可被操作。
focusform	指定在第一次显示屏幕时获得操作焦点的窗体实例。 选择性指定

属性	描述
preload	指定在加载对话框时是否预先加载并实例化下属屏幕。 true: 加载对话框时实例化屏幕。 false（缺省）: 在显示屏幕前才实例化屏幕。 选择性指定
terminate	指定在隐藏屏幕后是否摧毁屏幕。 true: 在隐藏屏幕后摧毁屏幕。 false（缺省）: 只有在摧毁对话框时才摧毁屏幕。 选择性指定

屏幕类

您可以通过实现一个屏幕类在 HMI 屏幕中插入自定义的功能。

每个 HMI 屏幕类必须由 GUI Framework 的 SIGfwHmiDialog 类导出。
SIGfwScreen 不仅提供导出的屏幕类的接口定义，而且提供基本实现。

通过改写 SIGfwScreen 的虚拟方法，屏幕可对特定事件作出响应。可用的方法参见本章末尾 SIGfwScreen 的接口引用。

! **重要提示**

在改写一个虚拟方法时，必须一同调用基本类的实现。

以下情形下最好实现一个屏幕类：

- 需要在显示屏幕时操作软键时，即激活或禁用软键。
- 需要向另一个屏幕传送数据或者要从另一屏幕接收数据（模态屏幕）时。
- 需要修改显示和隐藏屏幕时的属性时。

示例：

```
#ifndef MY SCREEN H INCLUDED
#define MY SCREEN H INCLUDED

#include "slgfwscreen.h"

class MyScreen :public SlGfwScreen
{
    Q OBJECT;

public:
    MyScreen(SlGfwHmiDialog* pParent = 0,
             const QString& rszName = QString::null);

    virtual ~MyScreen(void);

    virtual int init(const QString& rszArgs = QString::null);

    virtual void onFunction(const QString& rszFunction,
                            const QString& rszArgs,
                            bool& rbHandled);
};

#endif // MY_SCREEN_H_INCLUDED
```

在上例中可以看到屏幕类的头文件的内容。init()和 onFunction()这两个虚拟方法被改写。

插件宏命令

利用下述宏命令可以将该屏幕类通知给库中的插件机制：

SL_GFW_SCREEN_PLUGIN_EXPORT(<类名称>)

屏幕类的名称作为参数指定。

示例(plugin.cpp)：

```
#include "slgfwpluginmakros.h"
#include "myscreen.h"

SL GFW BEGIN PLUGIN EXPORT()
    SL_GFW_SCREEN_PLUGIN_EXPORT(MyScreen)
    // …更多导出的类
SL_GFW_END_PLUGIN_EXPORT()
```

4.5.1 模态屏幕

模态屏幕是指在该屏幕显示时会阻止在其他屏幕的可见窗体内输入数据。之前屏幕的窗体不会被隐藏，而是被锁定，即无法被操作。

组态

和非模态屏幕一样，模态屏幕是利用 XML 标签 SCREEN 来定义的。将属性 execmode 设为“modal”即可。

```
<SCREEN name="MyModalScreen" execmode="modal">
    ... 窗体定义
    … 菜单定义
</SCREEN>
```

模态屏幕中的窗体

模态屏幕中的窗体通常不会占据整个窗体区域，大部分位于窗体区域的中心，是小型的消息窗体或输入窗体。非模态屏幕的窗体通过蓝色外框和深色背景色突出显示。

标准屏幕布局中只提供有限的几个窗体大小，因此您通常需要自行设计项目特有的屏幕布局，在其中确定模态屏幕的窗体大小。此时可以利用 XML 属性 screenlayout 进行设计。

模态屏幕的显示与隐藏

模态屏幕有两种调用方式：

1) 非阻塞
和非模态屏幕一样，模态屏幕是通过方法 switchToScreen()显示的。这种调用是非阻塞调用，程序不被挂起，而是继续运行。结果从调用屏幕内诸如 open()方法内的其他位置上获得。因此在连续调用多个模态屏幕时，需要编程状态机以继续执行程序。

2) 阻塞
模态屏幕通过方法 execModalScreen()显示。与第 1 种调用方式的不同之处在于：此时程序一直被挂在调用代码行上，在退出模态屏幕后才继续运行。返回值（一个 QVariant）可以由 SIGfwScreen::setModalResult()设置。

模态屏幕通常通过调用方法 switchToDynamicTarget()隐藏。在模态屏幕显示时，调用屏幕将自己作为动态目标传送。对话框类利用方法 activeScreen()返回当时激活的屏幕的名称。

```
QString szArgs = QString("-slGfwDynamicScreen %1").arg(activeScreen());
switchToScreen("MyModalScreen", szArgs);
```

模态屏幕的显示流程为：

- 调用 witchToScreen()或 execModalScreen()以显示模态屏幕
- 锁定当前屏幕 (disable())
- 关闭当前屏幕(close())，其窗体保持可见
- 打开模态屏幕(open())
- 激活模态屏幕(enable())

调用屏幕可以通过 argument 字符串向模态屏幕传送参数，模态屏幕随后在 open()方法中分析该参数。

如上所述，模态屏幕通常是通过浏览到某个动态目标隐藏的。浏览可以通过模态屏幕的一个软键实现或者通过调用方法 switchToDynamicTarget()实现。您也可以直接跳转到另一个非模态屏幕。

模态屏幕的隐藏流程为：

- 调用 switchToDynamicTarget()以隐藏模态屏幕
- 锁定模态屏幕 (disable())
- 关闭模态屏幕(close())
- 打开之前的屏幕(open())
- 激活之前的屏幕(enable())

在调用方法 switchToDynamicTarget()时或者在 close()方法中，模态屏幕可以向 argument 字符串填入参数。该参数也可以是属性的值。之前的屏幕在 open 方法中收到该参数，在其中分析参数。

展示如何以非阻塞的方式调用模态屏幕以及如何在模态屏幕之间交换数据的示例位于“GUIFrameWork\SlExGuiModalDialog”的示例目录下。

展示如何以阻塞的方式调用模态屏幕的示例
位于“GUIFrameWork\SlExGuiModalDialog2”的示例目录中。

4.6 HMI 窗体

术语解释

窗体是屏幕内的显示单元。它用于输入输出对话框专有的数据。

窗体是对话框通用的，换句话说，一个窗体可用于多个屏幕。窗体实例通常通过其名称调用。需要一个窗体用于多个屏幕时，您必须在各个屏幕中实现相同的窗体、指定相同的窗体实例名称。

XML 标签 FORM

在对话框配置文件中，窗体是用 XML 标签 FORM 来定义的。
XML 标签 FORM 支持以下属性：

表 4-4：XML 标签 FORM

属性	描述
implementation	窗体的实现。实现的说明格式为： *library.class* *library*: 包含窗体实现的库。 *class*: SIGfwDialogForm 导出的类的名称。 未指定该属性时，程序运行期间会实例化基本类 SIGfwDialogForm。 选择性指定
name	窗体实例的名称
formpanel	屏幕布局中窗体面板的名称，窗体面板确定了屏幕上窗体的位置和大小。
x, y, width, height	指定窗体位置（左上方）和大小。 x：左上方 X 坐标 y:左上方 Y 坐标 width:窗体宽度 height:窗体高度 上述所有指定应针对屏幕分辨率 640x480 进行。只有在没有定义窗体面板时才需要进行指定。 选择性指定
helpdocument	指定本窗体的在线帮助文件。按下“INFO”键后并且本窗体具有操作焦点时，显示该文件的内容。 选择性指定
helpanchor	指定从哪个 HTML 锚点开始显示 *helpdocument* 属性指出的在线帮助文件。 选择性指定

属性	描述
visible	指定在第一次显示屏幕时是否要显示窗体。 true（缺省）： 在显示屏幕时显示窗体。 false: 在显示屏幕时不显示窗体。 选择性指定
zoom	指定在提高图像分辨率时是放大窗体及其内容还是只放大窗体。指定后者时，可通过提高图像分辨率来获得更大空间，显示更多信息。 true（缺省）：根据图形分辨率来调整窗体大小及其内容。 false:只根据图像分辨率调整窗体大小。 选择性指定
overlapping	指定显示窗体时如何处理已显示的窗体。 true:被覆盖的窗体保持可见 false（缺省）:被部分覆盖和完全覆盖的窗体被隐藏。设置了可调面板的窗体被缩小。 选择性指定
preload	指定在加载对话框时是否预先加载并实例化下属窗体。 true: 加载对话框时实例化窗体。 false（缺省）: 在显示窗体前才实例化窗体。 选择性指定
terminate	指定在隐藏窗体后是否摧毁窗体。 true: 在隐藏窗体后摧毁窗体。 false（缺省）: 只有在摧毁对话框时才摧毁窗体。 选择性指定

固定的显示尺寸

您可以在定义窗体时就确定窗体的大小。

为此您要在 XML 标签 FORM 中指定参数 x、y 和 width。

其中指定的大小是分辨率为 640x480 时的窗体大小。如果为 HMI 设置了其他分辨率，GUI Framework 会进行相应的换算。

一般来说，您最好通过窗体面板来指定窗体大小和位置。

窗体面板（Form panel）

您可以通过窗体面板来定义屏幕下属窗体的位置和大小。该窗体面板在定义屏幕布局时指定。

窗体面板利用屏幕布局文件中的的 XML 标签<FORMPANEL>定义。
XML 标签 FORM 支持以下属性：

表 4-5：XML 标签 FORMPANEL

属性	描述
id	窗体面板的 ID。 使用标准屏幕布局 SIStandardScreenLayout 或扩展屏幕布局时，您可以使用下述预定义的窗体面板： - “FullForm”，“FullFormAdaptable” - “LeftForm”，“RightForm” - “ModalForm”、“SmallModalForm”、“ModalFindForm”、“ModalReplaceForm”、“ModalNumberForm”和“ModalSettingForm” - “UpperForm”，“LowerForm” 不同分辨率的屏幕上窗体的位置和大小可参见文件。 ”`[Install-Dir]\hmisl\siemens\sinumerik\hmi\base\slstandardscreenlayout.xml`”。
x, y	窗体面板位于左上方。 x：左上方 X 坐标 y:左上方 Y 坐标
width	窗体面板的宽度。 没有指定窗体面板宽度时，宽度会自动根据水平方向上的空余位置进行调整。
height	窗体面板的高度。 没有指定窗体面板高度时，高度会自动根据垂直方向上的空余位置进行调整。

在屏幕配置文件中的 XML 属性 formpanel 中指定窗体面板的 ID，即可将该面板指定给窗体。

没有指定窗体面板的高度或宽度时，窗体会自动根据空余位置调整大小。随后显示多个窗体时，只要有可能，该窗体就会自动变小，为新窗体空出位置。隐藏了一些窗体时，该窗体又会根据现有空余位置自动变大。

窗体类

只有通过实现一个窗体类才可使窗体包含显示单元（小部件 widget）。在实现时可编写窗体针对不同事件的响应。

每个窗体类必须由 GUI Framework 的 SlGfwDialogForm 类导出。
SlGfwDialogForm 不仅提供导出的窗体类的接口定义，而且提供基本实现。

通过改写 SlGfwDialogForm 的虚拟方法，窗体可对特定事件作出响应。可用的方法详见 SlGfwDialogForm 的接口说明。

! **重要提示**

在改写一个虚拟方法时，必须一同调用基本类的实现。

示例：

```
#ifndef MY_FORM_H_INCLUDED
#define MY_FORM_H_INCLUDED

#include "slgfwdialogform.h"

class MyForm :public SlGfwDialogForm
{
    Q_OBJECT;

public:
    MyForm(QWidget* pParent = 0,
           const QString& rszName = QString::null);

    virtual ~MyForm(void);

    virtual void formInitialized(void);

    virtual void attachedToScreen(const QString& rszScreen);
};

#endif // MY_FORM_H_INCLUDED
```

在上例中可以看到窗体类的头文件的内容。formInitialized()和 attachedToScreen()这两个虚拟方法被改写。 formInitialized() 通知窗体已经初始化。attachedToScreen()通知窗体显示在一个屏幕中。

通过 C++编程设计小部件

如果一个窗体只有少数几个显示单元和操作单元（小部件），通常您可以通过 C++编程在窗体中生成这些小部件并将它们加入到窗体中。

窗体小部件在窗体的构造函数中生成。借助设置小部件的属性并在窗体中应用 Qt 布局，您可以修改窗体小部件的位置和大小。

示例：
下面的窗体会显示一个文本输入栏(Line Edit)和一个文本输出栏(Label)，这两栏由 Qt 布局水平对齐。

```
#ifndef MY_FORM_H
#define MY_FORM_H

#include "slgfwdialogform.h"

class QHBoxLayout;
class SlGfwLabel;
class SlGfwLineEdit;

class MyForm :public SlGfwDialogForm
{
    Q_OBJECT

public:
    MyForm(QWidget* pParent = 0,
           const QString& rszName = QString::null,
           Qt::WFlags f = 0);
    virtual ~MyForm(void);

private:
    QHBoxLayout* m_pLayout;
    SlGfwLabel* m_pLabel;
    SlGfwLineEdit* m_pLineEdit;

};

#endif // MY_FORM_H
```

这就是窗体的头文件。其中为小部件生成了三个私有的成员函数：Qt-Layout、Label 和 Line Edit。其中使用的类被申明为“forward”。

```
#include "myform.h"
#include <QHBoxLayout>
#include "slgfwlabel.h"
#include "slgfwlineedit.h"

MyForm::MyForm(QWidget* pParent, const QString& rszName,
               Qt::WFlags f)
    : SlGfwDialogForm(pParent, rszName, f)
{
    m_pLayout = new QHBoxLayout(this);

    m_pLabel = new SlGfwLabel(this);
    m_pLabel->setText("Ihre Eingabe: ");

    m_pLineEdit = new SlGfwLineEdit(this);

    m_pLayout->addWidget(m_pLabel);
    m_pLayout->addWidget(m_pLineEdit);
}

MyForm::~MyForm(void)
{
}
```

这就是窗体的 C++源文件。开头首先加入使用的小部件的头文件。之后在窗体的构造函数中生成小部件，最后将它插入到水平布局中。

由于构造函数中的所有小部件都获得了“parent”窗体，因您无需在窗体的析构函数中摧毁这些小部件。

通过 Qt Designer 设计小部件

Qt 公司提供有工具 Qt Designer，它用类似于 Visual Studio 的方式设计用户界面。Qt Designer 还可以用于确定小部件在 HMI 窗体内的位置。我们建议您采用外部软件 Qt Designer，而不是 Visual Studio 插件。

Qt Designer 将操作界面的设计保存在一份 UI 文件中。该文件由用户界面编译器 UIC 加以编译，生成一份头文件。该头文件包含了一个简单的数据类，它包含了指向窗体小部件的指针（成员函数）。窗体类从该数据类导出，进而“继承”了小部件。

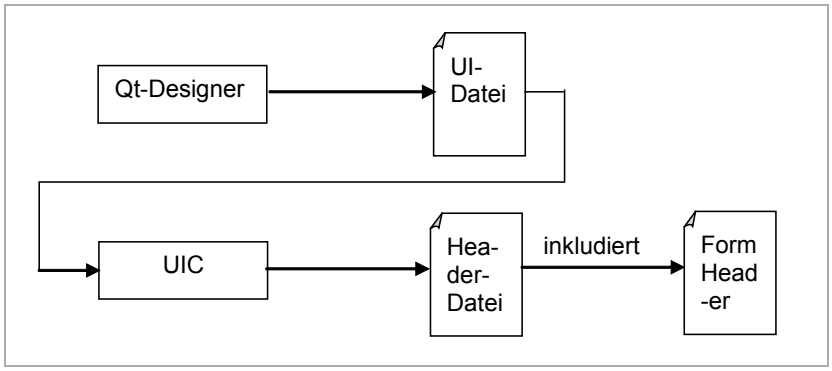


图 4-5:Qt Designer 的设计流程

点击开始菜单，启动 Qt Designer。在启动后 Qt Designer 会询问您使用哪个模板。HMI 窗体有六份模板，区别只在于窗体的大小。这六个窗体的大小是标准屏幕布局中分辨率为 640x480 时的标准窗体大小。

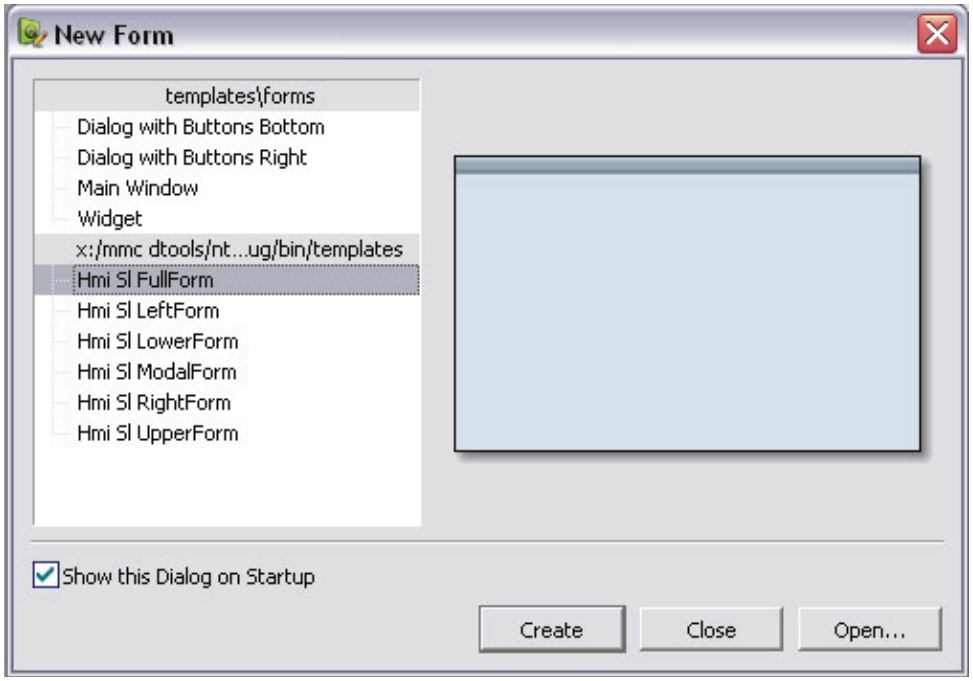


图 4-6:Qt Designer：选择窗体模板

按下按钮“Create”选择一个窗体模板。现在可以将小部件从“Widget”中拖动到该窗体上。 “Widget”的“HMI-Solutionline”栏下也有 HMI 小部件。

在“Property”编辑器中您可以设置窗体和小部件的属性。此时须注意：

- 窗体的属性“objectName”同时也是 UIC 生成的数据类的名称。此处最好使用相同的名称，之后的窗体类名称末尾可以加上一个 Ui(MyForm --> MyFormUi)。
- 小部件的属性“objectName”同时也是数据类中为小部件创建的成员函数的名称（指向该小部件的指针）。

已批准
Sheep , 19:56:43, 2015/07/21

关于如何使用 Qt Designer 的更多说明请参见 Qt 文档。

在 Qt Designer 中以 UI 文件的格式保存窗体设计，比如：myform.ui。

将该 UI 文件添加到您的项目中。

在 Visual Studio 中您必须为该 UI 文件设置 Custom Build Step，以便 UIC 可以编译该文件。 为此在 Solution Explorer 中右击 UI 文件，选择“Properties”。 在该窗口的左侧树形图中择“Custom Build Step”。在窗口右侧进行如下设置：
Command Line
$(QTDIR)\bin\uic $(InputPath) -o .\GeneratedFiles\ui_$(InputName).h
说明
Uic'ing $(InputFileName)...
Outputs
.\GeneratedFiles\ui_$(InputName).h

在窗体类的头文件中加入由 UIC 生成的数据类的头文件，比如：ui_myform.h。

除了从 SIGfwDialogForm 导出窗体类外，也要从数据类导出窗体类。 窗体类位于命名空间“Ui”内，名称为您在 Qt Designer 中指定的名称，比如：Ui::MyFormUi。从数据类中导出窗体类后，它也从数据类继承了小部件用作窗体的成员函数。

在窗体类的构造函数中调用方法 setupUi()，该方法从数据类继承，生成窗体小部件。

窗体类的头文件因此包含以下内容：

```
#ifndef MY_FORM_H
#define MY_FORM_H

#include "slgfwdialogform.h"
#include "ui_myform.h"

class MyForm :public SlGfwDialogForm,
               public Ui::MyFormUi
{
    Q_OBJECT

public:
    MyForm(QWidget* pParent = 0,
           const QString& rszName = QString::null,
           Qt::WFlags f = 0);
    virtual ~MyForm(void);

};

#endif // MY_FORM_H
```

窗体类的源文件内容为：

```
#include "myform.h"

MyForm::MyForm(QWidget* pParent, const QString& rszName,
               Qt::WFlags f)
    : SlGfwDialogForm(pParent, rszName, f)
{
    setupUi(this);
}

MyForm::~MyForm(void)
{
}
```

在调用 setupUi()后，会生成所有在 Qt Designer 中为窗体设计的小部件。窗体类从数据类继承了小部件的指针用作成员函数。该成员函数的名称和在 Qt Designer 中指定的小部件属性“objectName”一样。

标题栏和状态栏

每个 SIGfwDialogForm 都可以包含一个标题栏(CaptionBar)和一个状态栏(StatusBar)。这两栏可以包含多个部分（文字和图片）。下表列出了为设计提供的函数。括弧中的是 GET 函数，您可以据此检查进行了哪些设置。

表 4-6：标题栏/状态栏的函数

功能	描述
setCaptionBarVisible() (captionBarVisible())	指定是否显示标题栏/状态栏。 示例： `setCaptionBarVisible(true);` ➔ 标题栏现在可见。
setStatusBarVisible() (statusBarVisible())	
setCaptionBarElementCount() (captionBarElementCount())	指定一个标题栏/状态栏由多少个部分组成。 示例： `setStatusBarElementCount(3);` ➔ 状态栏由 3 部分组成
setStatusBarElementCount() (statusBarElementCount())	
setCaptionBarElementPicture() (captionBarElementPicture())	指定某部分的 QPixmap。 示例： `setCaptionBarElementPicture(&pixmap, 2);` ➔ pixmap 设为第 3 部分的图片。
setStatusBarElementPicture() (statusBarElementPicture())	
setCaptionBarElementText() (captionBarElementText())	指定某部分的文字。 示例： `setStatusBarElementText("Hallo", 0);` ➔ 查找第 1 部分的文字“Hallo”。
setStatusBarElementText() (statusBarElementText())	
setCaptionBarElementStartPosition() (captionBarElementStartPosition())	设置某部分的开始位置（%）。 示例： `setCaptionBarElementStartPosition(30, 1);` ➔ 第 2 部分从总长度的 30%处开始。
setStatusBarElementStartPosition() (statusBarElementStartPosition())	
setCaptionBarElementLength() (captionBarElementLength())	设置某部分的长度（%）。 示例： `setStatusBarElementLength(40, 0);` ➔ 第 1 部分占总长度的 40%。
setStatusBarElementLength() (captionBarElementLength())	
setCaptionBarElementAlignment() (captionBarElementAlignment())	设置某部分的对齐方式。 示例： `setCaptionBarElementAlignment (Qt::AlignCenter, 2);` ➔ 第 3 部分居中对齐。
setStatusBarElementAlignment() (statusBarElementAlignment())	

示例 1：设计一个由三部分组成的标题栏

```
void myForm::some fkt(...)
{
   ...
   QPixmap pixmap;
   ...
   setCaptionBarElementCount(3);
   ...
   setCaptionBarElementText("FirstText", 0);
   setCaptionBarElementStartPosition(0,  0);
   setCaptionBarElementLength(30, 0);
   setCaptionBarElementAlignment(Qt::AlignLeft, 0);

   setCaptionBarElementText("SecondText", 1);
   setCaptionBarElementStartPosition(35,  1);
   setCaptionBarElementLength(30, 1);
   setCaptionBarElementAlignment (Qt::AlignCenter, 1);

   setCaptionBarElementPicture(&pixmap,  2);
   setCaptionBarElementStartPosition(70,  2);
   setCaptionBarElementLength(30, 2);
   setCaptionBarElementAlignment(Qt::AlignRight, 2);
   ...
   setCaptionBarVisible(true);
}
```

本示例中标题栏由 3 部分组成：

- 第一部分是文字“FirstText”，左对齐，从最左边开始，长度最多占总长度的 30%。
- 第二部分是文字“SecondText”，居中对齐，从总长度 35%的位置开始，最多占总长度的 30%。
- 第三部分是图片“QPixmap”，右对齐，从总长度 70%的位置开始，最多占总长度的 30%。

示例 2：设计一条简单的状态栏

```
void myForm::some fkt(...)
{
   ...
   setStatusBarElementCount(1);
   setStatusBarElementAlignment(Qt::AlignCenter, 0);
   ...
   setStatusBarElementText("write some status information", 0);
   setStatusBarVisible(true);
}
```

本例是一个简单的状态栏设计，输出的文字“write some status information”居中。

提示框

需要为小部件自动显示提示框时，必须将该提示框告知窗体(addToToolTips)。提示框也可以即时显示(tipNow)。

表 4-7：SlGfwDialogForm 中用于提示框的方法

属性	描述
addToToolTips	将一个小部件加入到提示框列表中。一旦小部件获得焦点（或者鼠标在小部件上不动）而此后焦点没有继续移动（同时鼠标也不移动），则显示提示框。为获取提示框数据，调用 getToolTipInfo 。
removeFromToolTips	从提示框列表中删除小部件。
clearToolTips	删除提示框列表。
getToolTipInfo	需要显示某个小部件的提示框时，调用该属性（不适用于 ipNow！）。此处您可以修改缺省文本。
tipNow	没有延时地立即显示小部件的提示框。

重要方法

表 4-8：SlGfwDialogForm 的方法 - 通用

方法	描述
virtual formInitialized	在初始化窗体后被调用。从现在起可以访问对话框对象。
virtual attachedToScreen	在一个屏幕中显示窗体前被调用。软键已经显示在界面上，可以在此处加以修改。
virtual detachedFromScreen	在一个屏幕中隐藏窗体前被调用。
virtual languageChanged	在切换语言时被调用。最好重新设置窗体的所有文本。
readText	在读取语言文本时被调用。文本文件必须用 setTextResource 加载，再用 removeTextResource 卸载 。
virtual resolutionChanged	在切换分辨率时被调用。
virtual onAccessLevelChanged	在 NC 访问级别有变化时被调用。

插件宏命令

利用下述宏命令可以将窗体类通知给库中的插件机制：

SL_GFW_DIALOGFORM_PLUGIN_EXPORT(<类名称>)

窗体类的名称作为参数指定。

示例(plugin.cpp)：

```
#include "slgfwpluginmakros.h"
#include "myform.h"

SL_GFW_BEGIN_PLUGIN_EXPORT()
    SL_GFW_DIALOGFORM_PLUGIN_EXPORT(MyForm)
    // …更多导出的类
SL_GFW_END_PLUGIN_EXPORT()
```

4.7 HMI 小部件

编程包中有几个小部件已经开发好，定义了 HMI 的外观，以便您开发 HMI 窗体。

HMI 小部件类的使用需要以下头文件：

```
#include "slgfwlabel.h"
#include "slgfwlineedit.h"
#include "slgfwcombobox.h"
#include "slgfwtogglebox.h"
#include "slgfwradiobutton.h"
#include "slgfwcheckbox.h"
#include "slgrgrid.h"
```

4.7.1 修改

HMI 小部件中已经加入了几个属性，以实现 HMI 功能。

HMI 小部件的新属性

表 4-9：HMI 小部件的新属性

属性	类型	描述
hmiFonts	bool	设为 true 时，使用 SINUMERIK Operate 中设置的字体。随后会忽略所有 setFont 调用。方法 font()返回 HMI 自己的字体。 缺省值：true
hmiColors	bool	设为 true 时，使用 SINUMERIK Operate 中设置的颜色。随后会忽略所有 setPalette 调用。方法 palette()返回 HMI 自己的颜色。 缺省值：true
hmiUseBigFont	bool	设为 true 时，使用 HMI 中设为“大字体”的字体。 缺省值：false
hmiUseAsianFont	bool	设为 true 时，在必要时使用另一种字体，比如：亚洲字体。 设为 false 时一直使用 Latin1 字体。 缺省值：true

字体大小

为了避免不需要的反锯齿效果（Antialiasing），您最好只使用字体“Siemens AD Sans”
的下述大小（指像素值，而不是磅值）：
➔ 13px, 14px, 16px, 20px, 24px, 28px, 35px, 46px

这些字体大小的字符作为“位图”保存，不借助矢量信息在字体中生成。

4.7.2 等比光标控制

等比光标控制指在方向键被按下后输入焦点切换到等比数列中的下一个小部件中。返回键按照 tabulator 顺序浏览。

这种光标控制采用以下方式来搜索等比数列中的下一个小部件：

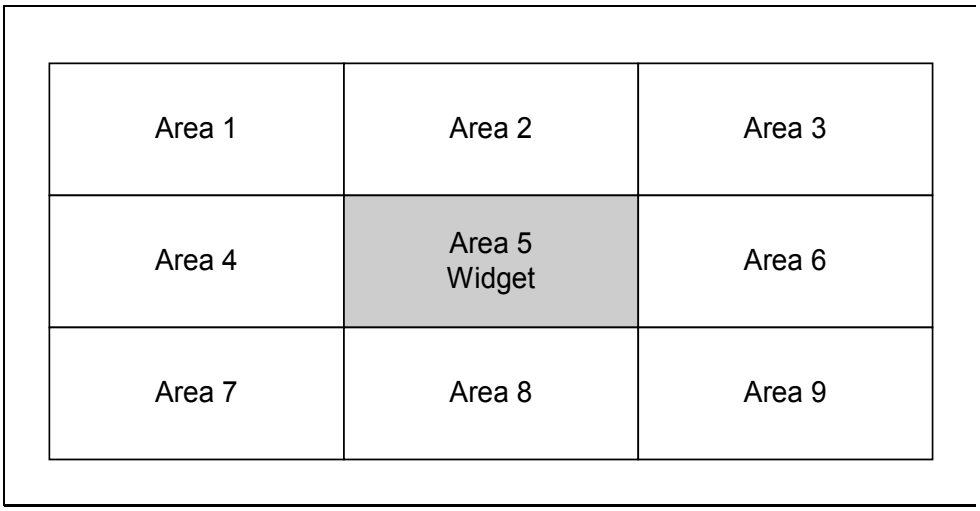


图 4-7:切换焦点时的搜索区域

按下↑键，区域的搜索顺序为：
区域 2 – 区域 1 – 区域 3
按下←键，区域的搜索顺序为：
区域 4 – 区域 1 – 区域 7
按下→键，区域的搜索顺序为：
区域 6 – 区域 3 – 区域 9
按下↓键，区域的搜索顺序为：
区域 8 – 区域 7 – 区域 9

下面两幅图反映了在按下↑↓←→键时小部件获得输入焦点的顺序。
比如，如左图所示，按下←键后，小部件 1 首先获得焦点。如果小部件 1 不允许获得焦点，则小部件 2 获得焦点，依此类推。

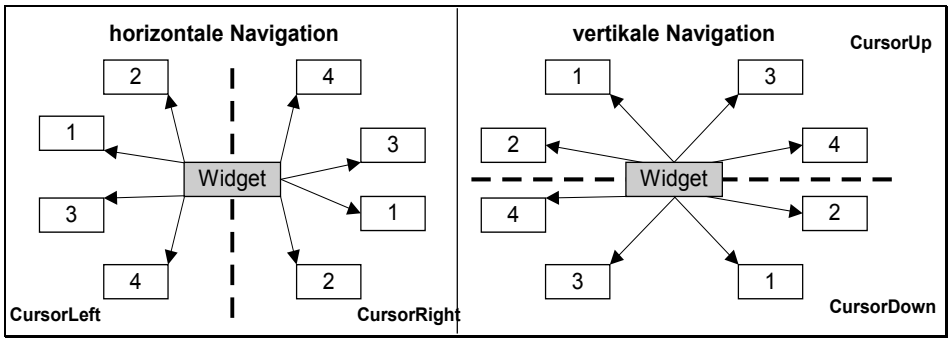


图 4-8:焦点切换顺序

如果在目标区域内有多个小部件，以按下↑键为例，输入焦点如下设置：

下面的这种格局对于等比光标控制来说是个问题：

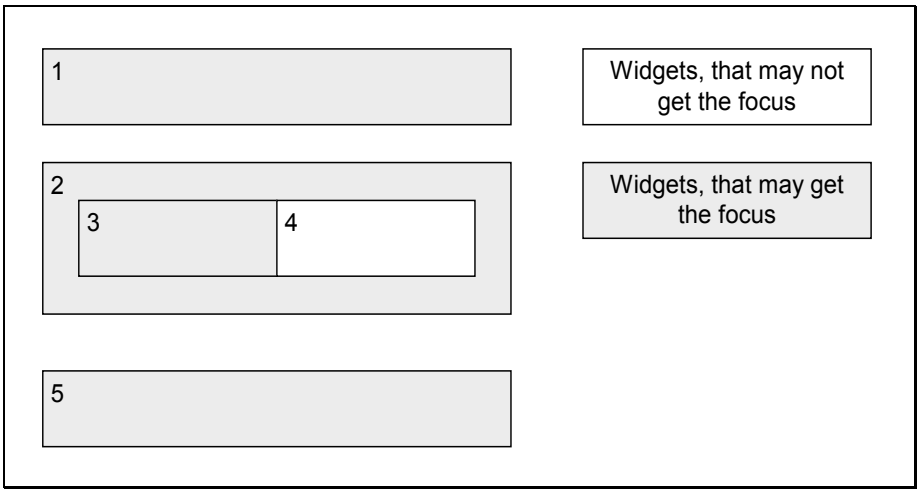


图 4-9:浏览时存在的问题

假设目前小部件 3 获得焦点，在按下↑键后应该按照上述顺序切换焦点：
3 -> 2 -> 1
然后再按下↓键继续切换焦点，此时应按照下述顺序切换：
1 -> 2 -> 5
也就是说，小部件 3 再也无法获得焦点。焦点总是从小部件 3 切换到小部件 2。
如果小部件 2 的 focusPolicy = NoFocus，就不会出现上述问题。此时小部件 2 用作 ContainerWidget，无法获得输入焦点。

关闭等比光标控制

如果您不希望使用等比光标控制，可以在窗体(SIGfwDialogForm)上调用以下函数：

```
setGeometricNavigationEnabled(false);
```

另外，您还可以在所有小部件上改写虚拟方法 `preTranslateNavigate()`。在需要进行浏览时都会调用该方法。如果那您不希望进行浏览，而希望单独处理方向键，将它设为 `false`。

4.7.3 浏览模式和编辑模式

某些小部件比如 SIGfwLineEdit 和 SIGfwComboBox 具有两种模式：浏览模式和编辑模式。小部件获得焦点时，一旦您按下“Insert”键或者开始输入，小部件便进入编辑模式。

在浏览模式中，方向键用于将焦点切换到下一个小部件。
在编辑模式中您可以修改小部件的数值。方向键不用于焦点切换。

表 4-10：SINUMERIK Operate 小部件的模式一览

小部件	焦点	是否允许输入	是否允许错误值
SIGfwLineEdit	始终可以获得焦点，焦点可通过以下函数关闭： setFocusPolicy() setDisabled() setEnabled()	是，可通过 setReadonly()设为只读	setErrorMode()
SIGfwComboBox		是	setErrorMode()
SIGfwCheckBox		是	否
SIGfwLabel		否	否
SIGfwRadioButton		是	否
SIGfwToggleBox		是	setErrorMode()

在通过浏览从编辑模式切换到浏览模式时，小部件有以下响应：

- 小部件将编辑模式设为 false。
- 发送信号 alueChanged()。
- 在输入错误时，信号接收方（比如：窗体）通过 setErrorMode()设置错误状态。
- 设置了错误状态后，浏览被中断。
- 只有没有设置错误状态时，才删除 undo 值。
- 在浏览模式中，方向键用于将焦点切换到下一个小部件。

4.7.4 SIGfwLabel

SIGfwLabel 是一个可以获得焦点的文本输出栏。

它和其它小部件一样，在浏览模式中，方向键用于切换焦点。

SIGfwLabel 默认为没有获得焦点 (focusPolicy = none)。

文本对齐：
可设置，默认为左对齐。

鼠标操作

表 4-11：SIGfwLabel 的鼠标操作

事件	之前的状态	响应
单击鼠标左键	没有获得焦点	该小部件现在获得焦点。处于浏览模式。
	浏览模式	没有响应。
双击鼠标左键	没有获得焦点	该小部件现在获得焦点。
	浏览模式	没有响应。
滚动鼠标滑轮	没有获得焦点	没有响应。
	浏览模式	没有响应。
单击鼠标右键	没有获得焦点	该小部件现在获得焦点。处于浏览模式。右键菜单打开。
	浏览模式	右键菜单打开。
双击鼠标右键		和单击鼠标右键的响应一样。

键盘操作

表 4-12：SIGfwLabel 的键盘操作

按键	操作
Ctrl + C	将小部件内当前包含的所有内容复制到剪贴板中。

接口

表 4-13：SIGfwLabel 的构造函数

	参数
SIGfwLabel	QWidget* pParent const QString& rszName = QString::null Qt::WFlags flags = 0
SIGfwLabel	const QString& rszText QWidget* pParent const QString& rszName = QString::null Qt::WFlags flags = 0

表 4-14：SlGfwLabel 的属性

格式/属性	读取/ 写入	描述
Qt::Alignment alignment	alignment setAlignment	文本的对齐方式
bool wordWrap	wordWrap setWordWrap	false:禁止自动换行 true:允许自动换行
int indent	indent setIndent	文本缩进 x 个像素
bool scaledContents	hasScaledContents setScaledContents	false:不整张显示图片 true:整张显示图片

表 4-15：SlGfwLabel 的信号

信号	参数	描述
gotFocus()	小部件获得了焦点	
	QWidget* pSource	发送信号的小部件
lostFocus()	小部件失去了焦点	
	QWidget* pSource	发送信号的小部件
mouseButtonPressed()	某个鼠标键被按下	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonClicked()	在同一个位置上单击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonDoubleClicked()	在同一个位置上双击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyDown()	某个按键被按下	
	int nKey	哪个按键被按下 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyUp()	某个按键被松开	
	int nKey	哪个按键被松开 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件

4.7.5 SIGfwLineEdit

SIGfwLineEdit 是一个占据一行的输入栏，有编辑模式和浏览模式。

在浏览模式中，方向键用于将焦点切换到下一个小部件。在编辑模式中，←→键用于在文本中移动光标。编辑模式和浏览模式之间的切换可通过输入字符、按下特殊快捷键、点击鼠标或切换焦点实现。

编辑模式有以下特点：

- 切换到浏览模式会触发 undo，输入栏内恢复进入编辑模式前的文本。
- 切换到编辑模式时（非鼠标操作），光标位于文本末尾。
- 通过编程接口设置文本时（见 setValue()），该文本不会显示在输入栏中，而是用作新的 undo 文本。
- 只有在编辑模式中标准的 undo/redo 机制（Ctrl+Z、Ctrl+Y）才有效。

表 4-16：SIGfwLineEdit 的文本对齐方式

	浏览模式	编辑模式
数字格式	右对齐	左对齐
文本	左对齐	左对齐

SIGfwLineEdit 支持多种输入模式，比如：整数值、浮点值等。输入模式可以通过 setInputMode()设置，见“接口”。

鼠标操作

表 4-17：SIGfwLineEdit 的鼠标操作

事件	之前的状态	响应
单击鼠标左键	没有获得焦点	该小部件现在获得焦点。整个文本被选中。处于浏览模式。
	浏览模式	切换到编辑模式，光标移动到由鼠标确定的位置。
	编辑模式	光标移动。
双击鼠标左键	没有获得焦点	该小部件现在获得焦点。整个文本被选中。处于浏览模式。
	浏览模式	切换到编辑模式，鼠标点击的字被选中。
	编辑模式	鼠标点击的字被选中。
滚动鼠标滑轮	没有获得焦点	没有响应。
	浏览模式	没有响应。
	编辑模式	没有响应。
单击鼠标右键	没有获得焦点	该小部件现在获得焦点。整个文本被选中。处于浏览模式。右键菜单打开。
	浏览模式	右键菜单打开。
	编辑模式	右键菜单打开。
双击鼠标右键		和单击鼠标右键的响应一样

键盘操作

表 4-18：SIGfwLineEdit 的键盘操作

按键	操作
←	**编辑模式**：光标向左移动一个字符。输入焦点不切换。
Shift + ←	**编辑模式**：光标向左移动一个字符并选中文本。输入焦点不切换。
→	**编辑模式**：光标向右移动一个字符。输入焦点不切换。
Shift + →	**编辑模式**：光标向右移动一个字符并选中文本。输入焦点不切换。
End	**编辑模式**：光标移动到行末。
Backspace	**浏览模式**：删除所有字符并切换到编辑模式 **编辑模式**：删除光标左侧的字符
Ctrl + Backspace	**浏览模式**：删除所有字符并切换到编辑模式 **编辑模式**：删除光标左侧的字
Delete	**浏览模式**：删除所有字符并切换到编辑模式 **编辑模式**：删除光标右侧的字符
Ctrl + Delete	**浏览模式**：删除所有字符并切换到编辑模式 **编辑模式**：删除光标右侧的字

按键	操作
Ctrl + C	将选中文本复制到剪贴板
Ctrl + V	将剪贴板中的文本粘贴到输入栏中。如果输入栏尚未设置，切换到编辑模式。
Ctrl + X	删除选中文本，将它复制到剪贴板中。如果输入栏尚未设置，切换到编辑模式。
Insert	在编辑模式和浏览模式之间来回切换（再次设置进入编辑模式前输入的数值）
Return	退出编辑模式

在编辑模式中，每次按下快捷键输入有效文本时，新文本会加入到小部件现有的文本中。在浏览模式中，输入的文本会改写现有文本。

计算器功能

在以下“InputMode”中会自动激活计算器：

- BINARY_MODE
- SIGNED_INTEGER_MODE
- UNSIGNED_INTEGER_MODE
- SIGNED_DOUBLE_MODE
- UNSIGNED_DOUBLE_MODE
- HEXADECIMAL_MODE
- DOUBLE_EXPONENT_MODE
- LENGTH_METRIC
- LENGTH_IMPERIAL
- LINEAR_FEED_PER_TIME_METRIC
- LINEAR_FEED_PER_TIME_IMPERIAL
- LINEAR_FEED_PER_REVOLUTION_METRIC
- LINEAR_FEED_PER_REVOLUTION_IMPERIAL
- LINEAR_FEED_PER_TOOTH_METRIC
- LINEAR_FEED_PER_TOOTH_IMPERIAL
- ANGLE
- ANGLE_INFEED
- REVOLUTION_SPEED
- CONSTANT_CUTTING_SPEED

在所有其他“InputMode”中可以通过“setCalculatorEnabled(true)”激活计算器。 此时还必须设置计算模式“setCalculatorMode”。

CalculatorModes:
CalcBinary = 0
CalcInteger = 1
CalcDouble = 2
CalcDoubleExp = 3
CalcHexadecimal = 4

此外还要将计算器屏幕加入到对话框定义中：

```
<IMPORT file="L:/gui/dialogs/common/slgfwcommon_incl.xml"
        screen="SlGfwCalculatorScreen" />
```

输入等号“**=**”即可显示计算器。

接口

注

在创建 SlGfWLineEdit 后我们建议设置 Inputmode。由此可以自动设置许多属性，无需您逐个设置。如果这些自动设置的属性不适用，您只需稍作修改。

表 4-19：SlGfwLineEdit 的构造函数

	参数
SlGfwLineEdit	QWidget* pParent = 0 const QString& rszName = QString::null
SlGfwLineEdit	const QString& rszContent QWidget* pParent = 0 const QString& rszName = QString::null
SlGfwLineEdit	const QString& rszContent const QString& rszInputMask QWidget* pParent = 0 const QString& rszName = QString::null

表 4-20：SlGfwLineEdit 的属性

格式/属性	读取/ 写入	描述
int maxLength	maxLength setMaxLength	输入文本的最大长度。超过该长度的文本会被截掉。 输入栏有一个 InputMask 时，长度由该标记设置。
EchoModeEnum echoMode	echoMode setEchoMode	LineEdit 的 Echomode 参见 EchoModeEnum
QString displayText	displayText	**Echomode NORMAL:** 文本同 text() **Echomode PASSWORD:** 密码以特定长度的星号显示。 **Echomode NO_ECHO:** 空字符串
int cursorPosition	cursorPosition setCursorPosition	光标位置

格式/属性	读取/ 写入	描述
Qt::Alignment alignment	alignment setAlignment	文本对齐 允许的值： Qt::AlignAuto Qt::AlignLeft Qt::AlignRight Qt::AlignHCenter

格式/属性	读取/ 写入	描述
bool modified	modified	缺省值：FALSE 输入栏的内容被修改时置为 TRUE。 在调用 **clearModified()** 或者 **setValue()** 时置为 FALSE
bool hasSelectedText	hasSelectedText	当输入栏中有文本被选中时置为 TRUE，否则为 FALSE
QString selectedText	selectedText	包含选中的文本。没有选中的文本时为 QString::null
bool readOnly	isReadOnly setReadOnly	TRUE:不允许编辑内容。 当 EchoMode 设为 NORMAL 时，允许复制和拖放内容。
bool acceptableInput	hasAcceptableInput	当输入和 Validator 以及 InputMask 相匹配时，置为 TRUE
bool calculatorOn	isCalculatorOn	显示了计算器时置为 TRUE
bool editMode	isEditMode	输入栏处于编辑模式时置为 TRUE
bool calculatorEnabled	isCalculatorEnabled setCalculatorEnabled	在数字格式中计算器启动。但是要定义屏幕。 输入等号“=”时计算器打开。在其他格式中必须设置 setCalculatorEnabled(true)。必须事先设置适合的 InputMode。
bool navigationOnReturn	navigationOnReturn setNavigationOnReturn	true:在编辑模式中按下“Return”时继续浏览。 false:在编辑模式中按下“Return”时终止浏览。小部件切换到浏览模式。
QString regExpInputValidator	regExpInputValidator setRegExpInputValidator	设置正则表达式输入验证（RegularExpressionInputValidator）。 由 InputMode 设置。
SIGfwDisplayModeEnum inputMode	inputMode setInputMode	设置/返回 InputMode 见下：InputMode
double minimumValue	minimumValue setMinimumValue	只在下述 inputMode 中有效的下限值： (UN)SIGNED_INTEGER_MODE (UN)SIGNED_DOUBLE_MODE (UN)SIGNED_EXPONENT_MODE 最大值和最小值相同时，取消限值检查。

格式/属性	读取/ 写入	描述
double maximumValue	maximumValue setMaximumValue	只在下述 inputMode 中有效的上限值： (UN)SIGNED_INTEGER_MODE (UN)SIGNED_DOUBLE_MODE (UN)SIGNED_EXPONENT_MODE 最大值和最小值相同时，取消限值检查。
int decimals	decimals setDecimals	小数点后的位数 只在下述 inputMode 中有效： (UN)SIGNED_DOUBLE_MODE (UN)SIGNED_EXPONENT_MODE 小于 0 的值被忽略。
bool leadingZeros	leadingZeros setLeadingZeros	TRUE:输入值以零开头，直到达到 maxLenght。
bool hideLastZeros	hideLastZeros setHideLastZeros	TRUE:删除小数点后的零。
bool disableCut	disableCut setDisableCut	TRUE:禁止剪切
bool enableInternalClaculator	enableInternalCalculator setEnableInternalCalculator	TRUE:激活内部计算器

表 4-21：SIGfwLineEdit 的信号

信号	参数	描述
textChanged()	文本被修改	
	const QString& rszNewText	新文本
	QWidget* pSource	发送信号的小部件
returnPressed()	“Return”或“Enter”键被按下。只有当输入符合 acceptableInput 时，才发出该信号。	
	QWidget* pSource	发送信号的小部件
selectionChanged()	选中的文本被修改	
	QWidget* pSource	发送信号的小部件
GotFocus()	小部件获得了焦点	
	QWidget* pSource	发送信号的小部件
lostFocus()	小部件失去了焦点	
	QWidget* pSource	发送信号的小部件

信号	参数	描述
mouseButtonPressed()	某个鼠标键被按下	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见： QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonClicked()	在同一个位置上单击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见： QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonDoubleClicked()	在同一个位置上双击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见： QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyDown()	某个按键被按下	
	int nKey	哪个按键被按下 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件

信号	参数	描述
keyUp()	某个按键被松开	
	int nKey	哪个按键被松开 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
valueChanged()	值被修改	
	QWidget* pSource	发送信号的小部件

EchoModeEnum

表 4-22：Enum EchoModeEnum

enum EchoModeEnum	描述
NORMAL	输入什么文本，便显示什么文本
NO_ECHO	不显示任何文本
PASSWORD	显示星号(*)而不是文本

InputMode

表 4-23：InputMode

模式	描述
ANY_MODE	全部字符
BINARY_MODE	仅 0 和 1 有效
SIGNED_INTEGER_MODE	仅有符号整数有效
UNSIGNED_INTEGER_MODE	仅无符号整数有效
SIGNED_DOUBLE_MODE	仅有符号双浮点数有效
UNSIGNED_DOUBLE_MODE	仅无符号双浮点数有效
HEXADECIMAL_MODE	仅十六进制数有效
ALPHA_MODE	仅字母有效
ALPHA_NUMERIC_MODE	仅数字和字母有效
PROGRAM_NAME_MODE	必须符合 (([A-Za-z]{2})|L|_)[A-Za-z0-9_]* */ 的规定
BLOCK_NUMBER_MODE	必须符合 [N:][0-9]* */ 的规定
DOUBLE_EXPONENT_MODE	仅含指数的有符号双浮点数有效
LENGTH_METRIC	“metric length axis”精度型双浮点数
LENGTH_IMPERIAL	“imperial length axis”精度型双浮点数
LINEAR_FEED_PER_TIME_METRIC	“metric feed per time axis”精度型双浮点数
LINEAR_FEED_PER_TIME_IMPERIAL	“imperial feed per time axis”精度型双浮点数
LINEAR_FEED_PER_REVOLUTION_METRIC	“metric feed per revolution axis”精度型双浮点数
LINEAR_FEED_PER_REVOLUTION_IMPERIAL	“imperial feed per revolution axis”精度型双浮点数
LINEAR_FEED_PER_TOOTH_METRIC	“metric feed per tooth axis”精度型双浮点数
LINEAR_FEED_PER_TOOTH_IMPERIAL	“imperial feed per tooth axis”精度型双浮点数
ANGLE	“angle”精度型双浮点数
ANGLE_INFEED	“infeed angle”精度型双浮点数
REVOLUTION_SPEED	“revolution speed”精度型双浮点数
CONSTANT_CUTTING_SPEED	“constant cutting speed”精度型双浮点数
BCD_CODED_VALUE	BCD 码格式的整数值
FILENAME_MODE	表示文件名称
IP_ADDRESS_MODE	表示 IPv4 地址，格式： 192.168.110.83

4.7.6 SIGfwComboBox

SIGfwComboBox 是由多个输入栏组成的下拉列表框。它分编辑模式和浏览模式。在编辑模式中可以看到一张下拉列表，在浏览模式中没有。

在浏览模式中，方向键用于将焦点切换到下一个小部件。在编辑模式中，←→键用于在文本中移动光标（可编辑的下拉列表框）。↑↓键用于在打开的下拉列表框中浏览下拉项。编辑模式和浏览模式之间的切换可通过输入字符、按下特殊快捷键、点击鼠标或切换焦点实现。

编辑模式有以下特点：

- 切换到浏览模式会触发 undo，输入栏内恢复进入编辑模式前的文本。
- 切换到编辑模式时（非鼠标操作），光标位于文本末尾（可编辑的下拉列表框）。
- 通过编程接口设置文本时（见 setValue()），该文本不会显示在输入栏中，而是用作新的 undo 文本。
- 只有在编辑模式中标准的 undo/redo 机制（Ctrl+Z、Ctrl+Y）才有效。

可编辑的下拉列表框指在输入文本期间也会显示下拉列表；在显示下拉列表同时也可在输入栏中编辑文本。输入文本时，如果它符合某个下拉项，则选中该下拉项。每个下拉项可以附加一个应用专用的数据(QVariant)。

表 4-24：SIGfwComboBox 的文本对齐

可编辑的下拉列表框	左对齐
不可编辑的下拉列表框	中心

鼠标操作

表 4-25：SIGfwComboBox 鼠标操作

事件	之前的状态	响应
单击鼠标左键	没有获得焦点	该小部件现在获得焦点。处于浏览模式。 **点击下拉列表框的按钮后**，下拉列表框切换到编辑模式，打开下拉列表。光标位于文本末尾。
	浏览模式	**可编辑的下拉列表框：**切换到编辑模式，下拉列表打开，当前文本被选中，显示在显示区。另外光标移动到由鼠标确定的位置。 **不可编辑的下拉列表框：**下拉列表打开，当前文本被选中 **点击下拉列表框的按钮后：**切换到编辑模式，下拉列表打开，当前文本显示在显示区。
	编辑模式	**可编辑的下拉列表框：** 见 SIGfwLineEdit。 **点击下拉列表框的按钮后：**下拉列表框切换到浏览模式，恢复在进入编辑模式前的最后有效值(undo)。 **点击某个下拉项：**立即选中该下拉项。下拉列表合上，该小部件切换到浏览模式。
双击鼠标左键	没有获得焦点	**可编辑的下拉列表框：**该小部件现在获得焦点。切换到浏览模式。 **不可编辑的下拉列表框：**该小部件现在获得焦点，下拉列表打开。 **点击下拉列表框的按钮后：**该小部件现在获得焦点。切换到浏览模式。

事件	之前的状态	响应
	浏览模式	**可编辑的下拉列表框：**切换到编辑模式，下拉列表打开，当前文本显示在显示区。另外，鼠标点击的字被选中。 **不可编辑的下拉列表框：**下拉列表打开，当前项被选中 **点击下拉列表框的按钮后：**仍处于浏览模式中
	编辑模式	**可编辑的下拉列表框：** 见 SIGfwLineEdit。 **点击下拉列表框的按钮后：**下拉列表合上，稍后再次打开。 **点击某个下拉项：** 该小部件切换到浏览模式，激活点中的下拉项

事件	之前的状态	响应
单击鼠标右键	没有获得焦点	该小部件现在获得焦点。处于浏览模式。 **可编辑的下拉列表框：** 见 SIGfwLineEdit。 **不可编辑的下拉列表框：**右键菜单打开。
	浏览模式	可编辑的下拉列表框：见 SIGfwLineEdit。 不可编辑的下拉列表框：右键菜单打开。
	编辑模式	可编辑的下拉列表框：见 SIGfwLineEdit。 不可编辑的下拉列表框：右键菜单打开。
双击鼠标右键		和单击鼠标右键的响应一样。

键盘操作

表 4-26：SIGfwComboBox 的键盘操作

按键	操作
←	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Shift + ←	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
→	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Shift + →	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
↑	**编辑模式：** 切换到上一条下拉项，但不会切换输入焦点！
↓	**编辑模式：** 切换到下一条下拉项，但不会切换输入焦点！
切换键	选中下一条下拉项。如果已达到列表末尾，则选中第一条下拉项。 当关闭了数码锁定 NumLock 时，操作面板上的切换键相当于数字键盘上的数字 5（也称 Clear 键）。
Shift + 切换键	选中之前的下拉项。如果已达到列表开头，则选中最后一条下拉项。 当关闭了数码锁定 NumLock 时，操作面板上的切换键相当于数字键盘上的数字 5（也称 Clear 键）。
End	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Backspace	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Ctrl + Backspace	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Delete	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Ctrl + Delete	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Ctrl + C	**编辑模式：** 将选中文本复制到剪贴板
Ctrl + V	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Ctrl + X	**编辑模式（可编辑的下拉列表框）：** 见 SIGfwLineEdit。
Insert	在编辑模式和浏览模式之间来回切换（再次设置进入编辑模式前输入的数值）
Esc	**编辑模式：**切换到浏览模式（再次设置进入编辑模式前输入的数值）

可编辑的下拉列表框：
见 SIGfwLineEdit。

不可编辑的下拉列表框：
每次按下快捷键输入有效且包含了下拉项的文本时，小部件会切换到编辑模式，选中该下拉项。

接口

表 4-27：SIGfwComboBox 的构造函数

	参数
SIGfwComboBox	QWidget* pParent = 0 const QString& rszName = QString::null

表 4-28：SIGfwComboBox 的属性

格式/属性	读取/ 写入	描述
bool editable	isEditable setEditable	用户可以编辑下拉列表框 缺省值：false
int count	count	当前下拉项的数量
QString currentText	currentText	当前下拉项的文本
int currentIndex	currentIndex setCurrentIndex	当前下拉项的索引 在下拉列表框为空或当前没有选中某个下拉项时返回-1。
int maxVisibleItems	maxVisibleItems setMaxVisibleItems	可显示的下拉项的最大数量
int maxCount	maxCount setMaxCount	下拉项的最大数量。如果该数量设得比当前下拉项的数量小，则多出的下拉项被剪切掉。
InsertPolicy insertPolicy	insertPolicy setInsertPolicy	指定何处要插入一条新的下拉项。 参见：enum InsertPolicy
SizeAdjustPolicy sizeAdjustPolicy	sizeAdjustPolicy setSizeAdjustPolicy	指定当插入了一条新的下拉项或者某个下拉项被修改时下拉列表框大小如何自动调整。 参见：enum SizeAdjustPolicy
int minimumContentsLenght	minimumContentsLenght setMinimumContentsLenght	缺省：0 一条下拉项包含的最少字符数。 没有设置 ADJUST_TO_MINIMUM_CONTENTS_LENGHT 时忽略此项。

格式/属性	读取/ 写入	描述
bool duplicatesEnabled	duplicatesEnabled setDuplicatesEnabled	TRUE:允许两条一样的下拉项。 FALSE:不允许两条一样的下拉项。 在编程时允许两条一样的下拉项。
bool navigationOnReturn	navigationOnReturn setNavigationOnReturn	true:在编辑模式中按下“Return”时继续浏览。 false:在编辑模式中按下“Return”时终止浏览。小部件切换到浏览模式。
bool editMode	isEditMode	下拉列表框处于编辑模式
QSize iconSize	iconSize setIconSize	在输入栏中显示的图标的大小 也设置 iconSizeList
QSize iconSizeList	iconSizeList setIconSizeList	在下拉列表中显示的图标的大小
bool extendListToRightSide	extendListToRightSide setExtendListToRightSide	下拉列表向右延长
Qt::Alignment listAlignment	listAlignment setListAlignment	下拉列表中文本的对齐方式

表 4-29：SIGfwComboBox 的信号

信号	参数	描述
GotFocus()	小部件获得了焦点	
	QWidget* pSource	发送信号的小部件
lostFocus()	小部件失去了焦点	
	QWidget* pSource	发送信号的小部件
mouseButtonPressed()	某个鼠标键被按下	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonClicked()	在同一个位置上单击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件

信号	参数	描述
mouseButtonDoubleClicked()	在同一个位置上双击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyDown()	某个按键被按下	
	int nKey	哪个按键被按下 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyUp()	某个按键被松开	
	int nKey	哪个按键被松开 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
valueChanged()	值被修改	
	QWidget* pSource	发送信号的小部件
editTextChanged()	下拉列表框中输入栏的文本被修改	
	const QString &text	新文本
	QWidget* pSource	发送信号的小部件
activated()	一条下拉项被选中	
	int nIndex	选中的下拉项的索引
	QWidget* pSource	发送信号的小部件
activated()	一条下拉项被选中	
	const QString &text	选中的下拉项的文本
	QWidget* pSource	发送信号的小部件

信号	参数	描述
highlighted()	一条下拉项高亮显示	
	int nIndex	高亮显示的下拉项的索引
	QWidget* pSource	发送信号的小部件
highlighted()	一条下拉项高亮显示	
	const QString &text	高亮显示的下拉项的文本
	QWidget* pSource	发送信号的小部件
currentIndexChanged()	当前索引被修改	
	int nIndex	新索引。 当下拉列表框为空或者 currentIndex 复位时返回-1。
	QWidget* pSource	发送信号的小部件
currentIndexChanged()	当前索引被修改	
	const QString &text	具有新索引的下拉项的文本
	QWidget* pSource	QWidget* pSource
returnPressed()	“Return”或“Enter”键被按下。只有当输入符合 acceptableInput 时，才发出该信号。	
	QWidget* pSource	发送信号的小部件

表 4-30：SIGfwComboBox 的槽

槽	参数	描述
clear()	删除所有下拉项	
clearEditText()	删除下拉列表框输入栏中的文本	
setEditText()	将下拉列表框输入栏中的文本设为传送的文本	
	const QString& rszText	需要设置的文本
setCurrentIndex()	设置当前索引	
	int nIndex	需要设置的索引

Enum InsertPolicy

表 4-31：enum InsertPolicy

enum InsertPolicy	描述
NO_INSERT	不插入字符串
INSERT_AT_TOP	字符串作为第一个下拉项插入
INSERT_AT_BOTTOM	字符串作为最后一个下拉项插入
INSERT_AFTER_CURRENT	字符串插入到当前下拉项后面
INSERT_BEFORE_CURRENT	字符串插入到当前下拉项前面

Enum SizeAdjustPolicy

表 4-32：enum SizeAdjustPolicy

enum SizeAdjustPolicy	描述
ADJUST_TO_CONTENTS	下拉列表框不自动调整其内容
ADJUST_TO_CONTENTS_ON_FIRST_SHOW	下拉列表框在第一次 show()时自动调整其内容
ADJUST_TO_MINIMUM_CONTENTS_LENGTH	下拉列表框自动根据 minimumContentsLength 进行调整

4.7.7 SlGfwToggleBox

SlGfwToggleBox 是一个下拉列表框。您可以切换下拉列表框中的数值。它基于 SlGfwComboBox，但是不可编辑，也没有按钮。

表 4-33：SlGfwToggleBox 的鼠标操作

事件	之前的状态	响应
单击鼠标左键	没有获得焦点	该小部件现在获得焦点。处于浏览模式。
	浏览模式	列表展开
双击鼠标左键	没有获得焦点	该小部件现在获得焦点，下拉列表展开
	浏览模式	列表展开
单击鼠标右键		右键菜单打开
双击鼠标右键		右键菜单打开

键盘操作

表 4-34：SlGfwToggleBox 的键盘操作

按键	操作
切换键	选中下一条下拉项。如果已达到列表末尾，则选中第一条下拉项。 当关闭了数码锁定 NumLock 时，操作面板上的切换键相当于数字键盘上的数字 5（也称 Clear 键）。
Shift + 切换键	选中之前的下拉项。如果已达到列表开头，则选中最后一条下拉项。 当关闭了数码锁定 NumLock 时，操作面板上的切换键相当于数字键盘上的数字 5（也称 Clear 键）。
Ctrl + C	将选中文本复制到剪贴板

接口

表 4-35：SlGfwToggelBox 的构造函数

	参数
SlGfwToggelBox	QWidget* pParent const QString& rszName = QString::null

4.7.8 SIGfwRadioButton

SIGfwRadioButton 在 QRadioButton 功能的基础上扩充的单选按钮。和 QRadioButton 不同的是，当鼠标点击 SIGfwRadioButton 文本/图标的四周时，也能进行切换。SIGfwRadioButton 默认获得焦点。

文本对齐：
左对齐

鼠标操作

表 4-36：: SIGfwRadioButton 的鼠标操作

事件	之前的状态	响应
单击鼠标左键	没有获得焦点	该小部件现在获得焦点，设置数值
	焦点	设置数值
双击鼠标左键		和单击鼠标左键的响应一样。
滚动鼠标滑轮		没有响应
单击鼠标右键	没有获得焦点	该小部件现在获得焦点。
	焦点	没有响应
双击鼠标右键		和单击鼠标右键的响应一样

键盘操作

表 4-37：SIGfwRadioButton 的键盘操作

按键	操作
切换键	设置数值 当关闭了数码锁定 NumLock 时，操作面板上的切换键相当于数字键盘上的数字 5（也称 Clear 键）。
空格键	设置数值

接口

表 4-38：SIGfwRadioButton 的构造函数

	参数
SIGfwRadioButton	QWidget* pParent const QString& rszName = QString::null
SIGfwRadioButton	const QString& rszText QWidget* pParent const QString& rszName = QString::null

表 4-39：SlGfwRadioButton 的属性

格式/属性	读取/ 写入	描述
Qt::Alignment iconAlignment	iconAlignment setIconAlignment	图标的对齐方式 允许的方式有： AlignAuto AlignLeft AlignRight AlignHCenter

表 4-40：SlGfwRadioButton 的信号

信号	参数	描述
GotFocus()	小部件获得了焦点	
	QWidget* pSource	发送信号的小部件
lostFocus()	小部件失去了焦点	
	QWidget* pSource	发送信号的小部件
mouseButtonPressed()	某个鼠标键被按下	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonClicked()	在同一个位置上单击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonDoubleClicked()	在同一个位置上双击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyDown()	某个按键被按下	
	int nKey	哪个按键被按下 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState

信号	参数	描述
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyUp()	某个按键被松开	
	int nKey	哪个按键被松开 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件

4.7.9 SIGfwCheckBox

SIGfwCheckBox 是在 QCheckBox 功能的基础上扩充的复选框。和 QCheckBox 不同的是，当鼠标点击 SIGfwCheckBox 文本/图标的四周时，也能进行切换。SIGfwCheckBox 默认获得焦点。

文本对齐：
左对齐

鼠标操作

表 4-41：SIGfwCheckBox 的鼠标操作

事件	之前的状态	响应
单击鼠标左键	没有获得焦点	该小部件现在获得焦点，切换数值
	焦点	切换数值
双击鼠标左键		和单击鼠标左键的响应一样。
滚动鼠标滑轮		没有响应
单击鼠标右键	没有获得焦点	该小部件现在获得焦点。
	焦点	没有响应
双击鼠标右键		和单击鼠标右键的响应一样

键盘操作

表 4-42：SIGfwCheckBox 的键盘操作

按键	操作
切换键	切换数值 当关闭了数码锁定 NumLock 时，操作面板上的切换键相当于数字键盘上的数字 5（也称 Clear 键）。
空格键	切换数值

接口

表 4-43：SlGfwCheckBox 的构造函数

	参数
SlGfwCheckBox	QWidget* pParent const QString& rszName = QString::null
SlGfwCheckBox	const QString& rszText QWidget* pParent = 0 const QString& rszName = QString::null

表 4-44：SlGfwCheckBox 的接口

格式/属性	读取/ 写入	描述
Qt::Alignment iconAlignment	iconAlignment setIconAlignment	图标的对齐方式 允许的方式有： AlignAuto, AlignLeft AlignRight, AlignHCenter

表 4-45：SlGfwCheckBox 的信号

信号	参数	描述
GotFocus()	小部件获得了焦点	
	QWidget* pSource	发送信号的小部件
lostFocus()	小部件失去了焦点	
	QWidget* pSource	发送信号的小部件
mouseButtonPressed()	某个鼠标键被按下	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonClicked()	在同一个位置上单击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
mouseButtonDoubleClicked()	在同一个位置上双击并松开了鼠标键。	
	int nButton	哪个鼠标键被按下 参见：Qt::ButtonState
	const QPoint& rPoint	全局鼠标坐标 参见：QMouseEvent::globalPos()
	rbIgnoreEvent	设为 true 时单击鼠标后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件

信号	参数	描述
keyDown()	某个按键被按下	
	int nKey	哪个按键被按下 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件
keyUp()	某个按键被松开	
	int nKey	哪个按键被松开 参见：Qt::Key
	int nButtonState	特殊按键，如 Shift、Ctrl 和 Alt 参见：Qt::ButtonState
	rbIgnoreEvent	设为 true 时按下按键后发生的所有后续事件都被忽略。
	QWidget* pSource	发送信号的小部件

4.7.10 SlGrGrid

SlGrGrid 用于设置基于单元格的表格视图。

表格的行与列由单个的单元格组成。每个单元格都由固定的元素占用。此处的元素可以是：

- SlGrGridItem ➔ 标准数据区（文本、数据、pixmap)
- SlGrCheckGridItem ➔ 复选框元素
- SlGrComboGridItem➔ 下拉列表框元素
- SlGrToggleGridItem➔ 切换框元素

SlGrGrid 可对垂直和水平的表格标题（表头）、所有元素及其选择进行操作处理。直接编辑文本或数据的功能也可被激活。

SINUMERIK Operate 编程包中的示例“SlExWidgetGrid”介绍了最常用的一些功能。

鼠标操作

表 4-46：SlGrGrid 的鼠标操作

事件	响应	
单击鼠标左键	通用： - 如之前 SlGrGrid 未获得焦点，则会置入焦点。 - 被点击选中的元素变为当前元素，小部件处于浏览模式下。 - 所作的选择被取消。 - 点击列标题，当属性“sorting 被设为 true 时，对该列所有单元格中的元素文本进行排序。 此外各个元素类型还分别适用：	
	SlGrGridItem	**不可编辑：** 无其他操作。 **可编辑：** 如果点击选中的元素就是当前元素，则直接切换到编辑模式。 如果元素已经处于编辑模式下，则光标会移动到相应位置。
	SlGrCheckGridItem	切换数值。
	SlGrComboGridItem	**不可编辑/可编辑：** 当点击下拉列表框的按钮时或者当点击选中的元素就是当前元素时，列表会打开。 如果点击列表元素将其选中，则列表会被关闭，小部件切换到浏览模式。 **只可编辑：** 另外光标移动到由鼠标确定的位置。 如果元素已经处于编辑模式下，则光标会移动到相应位置。

事件	响应	
		如果在编辑模式下点击下拉列表框的按钮，则会切换到浏览模式下，并且 恢复在进入编辑模式前的最后有效值(undo)。
	SlGrToggleGridItem	如果点击选中的元素就是当前元素，则列表打开。 如果点击列表元素将其选中，则列表会被关闭，小部件切换到浏览模式。
双击鼠标左键	通用： - 如之前 SlGrGrid 未获得焦点，则会置入焦点。 - 点击选中的元素变为当前元素。 - 如在表格标题区域中点击表格的分隔线，则可分别将行高或列宽设为最小值。 此外各个元素类型还分别适用：	
	SlGrGridItem	**不可编辑：** 无其他操作。 **可编辑：** 直接切换到编辑模式。 如果元素已经处于编辑模式下，则会选中光标下的字。
	SlGrCheckGridItem	切换数值。
	SlGrComboGridItem	**不可编辑：** 列表打开。 如点击下拉列表框的按钮，则小部件保持处于浏览模式。 **可编辑：** 小部件保持处于浏览模式。 如果元素已经处于编辑模式下，则会选中光标下的字。
	SlGrToggleGridItem	列表打开。
滚动鼠标滑轮	当表格各行未全部显示时，就进行垂直滚动。	
单击鼠标右键 双击鼠标右键	- 如之前 SlGrGrid 未获得焦点，则会置入焦点。 - 被点击选中的元素变为当前元素，小部件处于浏览模式下。 - 对于 SlGrGridItem、SlGrComboGridItem 和 SlGrToggleGridItem 将打开右键菜单。	
按住鼠标左键并移动	在表格标题区域中拖动表格的分隔线，可改变所选行或列的行高或列宽。	
Ctrl + 按住鼠标左键并移动	选中表格标题区域并拖动，可将所选行或列移动到需要的位置。中间的行或列会自动替补。 前提是属性“rowMovingEnabled”或“columnMovingEnabled”被设为 true。	

事件	响应
Shift + 单击鼠标左键	如果选择模式设为 MULTI_ROW，则可从当前行一直选中到所点击的行。
Ctrl + 单击鼠标左键	如果选择模式设为 MULTI_ROW，则可选中当前行和所点击的行。

键盘操作

表 4-47：SIGrGrid 的键盘操作

按键	操作
←	光标向左移动到该行中下一个可获得焦点的单元格。
→	光标向右移动到该行中下一个可获得焦点的单元格。
↑	光标向上移动到该列中下一个可获得焦点的单元格。
↓	光标向下移动到该列中下一个可获得焦点的单元格。
Ctrl + ←	如未显示表格的左边界，则可向左滚动一列。光标此时不移动。
Ctrl + →	如未显示表格的右边界，则可向右滚动一列。光标此时不移动。
Ctrl + ↑	如未显示表格的上边界，则可向上滚动一行。光标此时不移动。
Ctrl + ↓	如未显示表格的下边界，则可向下滚动一行。光标此时不移动。
Shift + ↑	光标向上移动到该列中下一个可获得焦点的单元格。 如果选择模式设为 MULTI_ROW，则可从当前行一直选中到光标向上移动到的行。
Shift + ↓	光标向下移动到该列中下一个可获得焦点的单元格。 如果选择模式设为 MULTI_ROW，则可从当前行一直选中到光标向下移动到的行。
Ctrl + Home	光标移动到该列中最上面一个可获得焦点的单元格。保持选中。
Shift + Ctrl + Home	光标移动到该列中最上面一个可获得焦点的单元格。 如果选择模式设为 MULTI_ROW，则可从当前行一直选中到识别到的最上面一个可获得焦点的单元格所在的行。
End	光标移动到该行中最右侧的可获得焦点的单元格。
Ctrl + End	光标移动到该列中最下面一个可获得焦点的单元格。
Shift + Ctrl + End	光标移动到该列中最下面一个可获得焦点的单元格。 如果选择模式设为 MULTI_ROW，则可从当前行一直选中到识别到的最下面一个可获得焦点的单元格所在的行。
Page Up	在一列中向上翻页。如不能再翻页，光标将置于第一个可显示的行中。保持选中。
Page Down	在一列中向下翻页。如不能再翻页，光标将置于最后一个可显示的行中。保持选中。
Shift + Page Up	在一列中向上翻页。如不能再翻页，光标将置于第一个可显示的行中。保持选中。 如果选择模式设为 MULTI_ROW，则可从当前行一直选中到向上可显示到的页面。
Shift + Page Down	在一列中向下翻页。如不能再翻页，光标将置于最后一个可显示的行中。保持选中。 如果选择模式设为 MULTI_ROW，则可从当前行一直选中到向下可显

按键	操作	
	示到的页面。	
Return	退出编辑模式并使光标在整个表格中移动，无需退出表格。 此时的顺序通过属性“orientationEnter”确定。取消选中。	
切换键	SlGrCheckGridItem	切换数值。
	SlGrComboGridItem SlGrToggleGridItem	选中列表的下一个元素。
Ctrl + 切换键	如果选择模式设为 MULTI_ROW，则所选行的模式可在已选择和未选择之间切换。	
Insert	取消选中。如果元素可编辑，则切换到编辑模式。	
Space	SlGrCheckGridItem	切换数值。
	SlGrComboGridItem SlGrToggleGridItem	列表打开。
Ctrl + c	将选中的单元格文本复制到剪贴板。	
Ctrl + x	将选中的单元格文本复制到剪贴板并删掉原来的单元格文本。	
Ctrl + v	用剪贴板中的文本覆盖选中的单元格文本。	

右键菜单

取决于各个单元格的元素类型和可编辑性，可能只会显示部分的右键菜单。

表 4-48：SlGrGrid 的右键菜单

。	操作
剪切	将选中的单元格文本复制到剪贴板并删掉原来的单元格文本。
复制	将选中的单元格文本复制到剪贴板。
插入	用剪贴板中的文本覆盖选中的单元格文本。

接口

表 4-49：SlGrGrid 的构造函数

	参数
SlGrGrid	int numRows int numCols QWidget* pParent = 0 const QString& rszName = QString::null

表 4-50：SlGrGrid 的属性

属性	读取/写入	描述
int numRows	numRows setNumRows	表格中的行数。
int numCols	numCols setNumCols	表格中的列数。
bool showGrid	showGrid setShowGrid	TRUE: 显示表格栅格。
bool rowMovingEnabled	rowMovingEnabled setRowMovingEnabled	TRUE: 在表格标题区域中“单击鼠标左键”并同时按下“Ctrl”将选中的行移动到所需位置。中间的行会自动替补。
bool columnMovingEnabled	columnMovingEnabled setColumnMovingEnabled	TRUE: 在表格标题区域中“单击鼠标左键”并同时按下“Ctrl”将选中的列移动到所需位置。中间的列会自动替补。
bool readOnly	isReadOnly	TRUE:

属性	读取/ 写入	描述
	setReadOnly	不允许编辑内容。
bool sorting	sorting setSorting	TRUE: 在列标题上“单击鼠标左键”对该列所有单元格的元素文本进行排序。
SortModeEnum sortMode	sortMode setSortMode	确定排序方式： SORT_ONLY_IN_COLUMN ➔只对所选列进行排序。排序不对其他列产生影响。 SORT_WHOLE_ROWS ➔对所选列进行排序。其他列也随之以相同顺序加以排序。 SORT_WHOLE_ROWS_AND_HEADER ➔与 SORT_WHOLE_ROWS 相同，但会额外将行标题加以排序。
SelectionModeEnum selectionMode	selectionMode setSelectionMode	确定选中方式： SINGLE_ROW ➔选中区域只为一行。 MULTI_ROW ➔总共可选中多个不同的区域。每个区域可包含多行。 NO_SELECTION ➔无法选择。
int numSelections	numSelections	表格中选中的数量。
bool currRowMark	isCurrRowMark setCurrRowMark	如果选中方式设为 MULTI_ROW，则选中(TRUE)或取消选中(FALSE)当前行。
OrientationEnterEnum orientationEnter	isOrientationEnter setOrientationEnter	确定按下 Return 键后哪个单元格获得焦点： 缺省值： FIRST_IN_ROW_RIGHT_THEN_NEXT_ROW_DOWN ➔首先光标在一行中从左至右移动。光标运行到一行末尾处将跳至下一行的最左侧。 类似的光标移动方式还有： FIRST_IN_ROW_LEFT_THEN_NEXT_ROW_DOWN FIRST_IN_ROW_RIGHT_THEN_NEXT_ROW_UP FIRST_IN_ROW_LEFT_THEN_NEXT_ROW_UP FIRST_IN_COLUMN_DOWN_THEN_NEXT_COLUMN_LEFT FIRST_IN_COLUMN_UP_THEN_NEXT_COLUMN_LEFT FIRST_IN_COLUMN_DOWN_THEN_NEXT_COLUMN_RIGHT FIRST_IN_COLUMN_UP_THEN_NEXT_COLUMN_RIGHT
int rowDistance	rowDistance setRowDistance	确定两行之间的间距。 间距为行高的 组成部分。
int columnDistance	columnDistance setColumnDistance	确定两列之间的间距。 间距为列宽的组成部分。

属性	读取/ 写入	描述
bool multicellularAllowed	isMulticellularAllowed setMulticellularAllowed	TRUE: 如果显示时单元格的文本过大，则单元格的高度会相应作出调整。

表 4-51：SlGrGrid 的信号

信号	参数	描述
gotFocus()	当表格获得焦点时，触发该信号。	
	QWidget* pSource	信号源
lostFocus()	当表格失去焦点时，触发该信号。	
	QWidget* pSource	信号源
mouseButtonPressed()	当按下鼠标键时，触发该信号。	
	int nButton	被按下的鼠标键 (见 Qt::ButtonState)
	const QPoint& rPoint	相对于整个小部件 SlGrGrid 左上角的鼠标坐标，考虑标题区和滚动区。
	int nRow	对应鼠标坐标的行下标。 (nRow=-1 表头)
	int nColumn	对应鼠标坐标的列下标。 (nColumn=-1 表头)
	bool& rbIgnoreEvent	如果该参数设为 TRUE，则不处理 SlGrGrid 小部件的事件。
	QWidget* pSource	信号源
mouseButtonClicked()	当单击（按下又松开）鼠标键时，触发该信号。	
	int nButton	被按下的鼠标键 (见 Qt::ButtonState)
	const QPoint& rPoint	相对于整个小部件 SlGrGrid 左上角的鼠标坐标，考虑标题区和滚动区。
	int nRow	对应鼠标坐标的行下标。 (nRow=-1 表头)
	int nColumn	对应鼠标坐标的列下标。 (nColumn=-1 表头)
	bool& rbIgnoreEvent	如果该参数设为 TRUE，则不处理 SlGrGrid 小部件的事件。
	QWidget* pSource	信号源
mouseButtonDoubleClicked()	当双击（两次按下又松开）鼠标键时，触发该信号。	
	int nButton	被按下的鼠标键 (见 Qt::ButtonState)
	const QPoint& rPoint	相对于整个小部件 SlGrGrid 左上角的鼠标坐标，考虑标题区和滚动区。
	int nRow	对应鼠标坐标的行下标。 (nRow=-1 表头)
	int nColumn	对应鼠标坐标的列下标。 (nColumn=-1 表头)
	bool& rbIgnoreEvent	如果该参数设为 TRUE，则不处理 SlGrGrid 小部件的事件。
	QWidget* pSource	信号源

信号	参数	描述
keyDown()	当按下按键时，触发该信号。	
	int nKey	哪个按键被按下。 (见 Qt::Key)
	int nButtonState	特殊按键，如 Shift、Ctrl、Alt 等 (见 Qt::ButtonState)
	bool& rbIgnoreEvent	如果该参数设为 TRUE，则不处理 SlGrGrid 小部件的事件。
	QWidget* pSource	信号源
keyUp()	当松开被按下的按键时，触发该信号。	
	int nKey	哪个按键被松开。 (见 Qt::Key)
	int nButtonState	特殊按键，如 Shift、Ctrl、Alt 等 (见 Qt::ButtonState)
	bool& rbIgnoreEvent	如果该参数设为 TRUE，则不处理 SlGrGrid 小部件的事件。
	QWidget* pSource	信号源
currentChanged()	当前单元格被修改。	
	int nRow	新的、当前行。
	int nCol	新的、当前列。
focusAreaInWidgetChanged()	当焦点在表格中变化时，发送该信号。	
	QWidget* pSource	信号源
visibleAreaChanged()	当表格的可见区域变化时，发送该信号。当随着滚动显示新区域之前或者使用光标键移动到之前不可见的单元格时，会立即发送信号。	
selectionChanged()	选中的内容变化时，发送该信号。	
valueChanged()	单元格中的值变化时，发送该信号。	
	int nRow	发生变化的单元格的行下标。
	int nCol	发生变化的单元格的列下标。
contextMenuRequested()	调用右键菜单（单击鼠标右键）时，触发该信号。	
	int nRow	调用了右键菜单的单元格的行下标。
	int nCol	调用了右键菜单的单元格的列下标。
	const QPoint& rPos	指定显示右键菜单的位置。

函数 - 管理数据

表 4-52：***setItem / item***

setItem: 在指定位置添加一个表格元素。已有数据此时会被覆盖。 ***item:*** 返回指定位置的表格元素。如该单元格不存在或未设置内容，则返回 0。	
void SlGrGrid::setItem(int nRow, int nCol, SlGrGridItem* pItem); **SlGrGridItem* SlGrGrid::item(int nRow, int nCol) const;**	
参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
pItem	单个表格元素 允许使用下列数据类型： ➔ SlGrGridItem, SlGrComboGridItem, SlGrToggleGridItem, SlGrCheckGridItem

表 4-53：***setValue / value***

setValue: 设置指定位置上表格元素的值。已有数据此时会被覆盖。如果该指定位置上还不存在表格元素(SlGrGridItem)，则会创建一个可编辑的 SlGrGridItem。 ***value:*** 返回指定位置上表格元素的值。	
void SlGrGrid::setValue(int nRow, int nCol, const QVariant& varValue, bool bSetUndo = true); **QVariant SlGrGrid::value(int nRow, int nCol) const;**	
参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
varValue	待设置的值
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-54：***setText / text***

setText: 设置指定位置上表格元素的文本。已有数据此时会被覆盖。如果该指定位置上还不存在表格元素(SlGrGridItem)，则会创建一个可编辑的 SlGrGridItem。 ***text:*** 返回指定位置上表格元素的文本。如果表格元素不含任何文本或不存在，则返回 QString::null。	
void SlGrGrid::setText(int nRow, int nCol, const QString& rText, bool bSetUndo = true); **QString SlGrGrid::text(int nRow, int nCol) const;**	
参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
rText	待设置的文本
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-55：***setPixmap / pixmap***

setPixmap: 设置指定位置上表格元素的 pixmap。已有数据此时会被覆盖。如果该指定位置上还不存在表格元素(SlGrGridItem)，则会创建一个可编辑的 SlGrGridItem。 ***pixmap:*** 返回指定位置上表格元素的 pixmap。如果表格元素不含 pixmap 或不存在，则返回一个空的 pixmap。
void SlGrGrid::setPixmap(int nRow, int nCol, const QPixmap& rPix, Qt::Alignment nAlign = Qt::AlignHCenter); **QPixmap SlGrGrid::pixmap(int nRow, int nCol) const;**

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
rPix	待设置的 pixmap
nAlign	pixmap 的对齐。允许的方式有： Qt::AlignLeft, Qt::AlignRight Qt::AlignTop, Qt::AlignBottom Qt::AlignVCenter, Qt::AlignHCenter

表 4-56：***clearCell***

清除指定表格元素的内容。
void SlGrGrid::clearCell(int nRow, int nCol);

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标

表 4-57：***insertRows / insertColumns***

insertRows: 从某一行起添加空的行。 ***insertColumns:*** 从某一列起添加空的列。
void SlGrGrid::insertRows(int nRow, int nCount = 1); **void SlGrGrid::insertColumns(int nCol, int nCount = 1);**

参数	含义
nRow	行
nCol	列
nCount	要添加的行数(insertRows)或 列数(insertColumns)。

表 4-58：***removeRow / removeColumn***

removeRow: 删除一整行。 ***removeColumn:*** 删除一整列。
void SlGrGrid::removeRow(int nRow); **void SlGrGrid::removeColumn(int nCol);**

参数	含义
nRow	要删除的行
nCol	要删除的列

表 4-59：***removeRows / removeColumns***

removeRows: 删除所有列出的行。 ***removeColumns:*** 删除所有列出的列。 提示!!! 列表(QVector)只允许包含有效行或列，不得重复且必须升序排列。
void SlGrGrid::removeRows(const QVector<int>& rRows); **void SlGrGrid::removeColumns(const QVector<int>& rCols);**

参数	含义
rRows	要删除的行的列表
rCols	要删除的列的列表

表 4-60：***copyRow / copyColumn***

copyRow: 复制一整行。 ***copyColumn:*** 复制一整列。
bool SlGrGrid::copyRow(int nRowDestination, int nRowSource, bool bConvert, bool bWithHeader); **bool SlGrGrid::copyColumn(int nColumnDestination, int nColumnSource, bool bConvert, bool bWithHeader);**

参数	含义
nRowDestination	要复制的行的目标位置
nRowSource	要复制的行的下标
nColumnDestination	要复制的列的目标位置
nColumnSource	要复制的列的下标
bConvert	TRUE: 复制所有单元格，而不比较单个单元格的元素类型（SlGrGridItem, SlGrCheckGridItem,…）。 FALSE: 只有源单元格与目标单元格的元素类型一致时，才进行复制。
bWithHeader	TRUE: 行标题或列标题也进行复制。 FALSE: 表格标题不进行复制。
返回值	复制任务的执行状态。

表 4-61：***swapRows / swapColumns***

swapRows: 交换完整的两行。 ***swapColumns:*** 交换完整的两列。 （接着应（例如）使用 updateContents()更新 SlGrGrid）
void SlGrGrid::swapRows(int nRow1, int nRow2, bool bSwapHeader = false); **void SlGrGrid::swapColumns(int nCol1, int nCol2, bool bSwapHeader = false);**

参数	含义
nRow1, nRow2	要交换的行
nCol1, nCol2	要交换的列
bSwapHeader	TRUE: 行标题或列标题也进行交换。 FALSE: 表格标题保持不变。

表 4-62：***swapCells***

交换两个单独的表格元素。 （接着应（例如）使用 updateContents()更新 SlGrGrid）
void SlGrGrid::swapCells(int nRow1, int nCol1, int nRow2, int nCol2);

参数	含义
nRow1, nCol1	要交换的第一个表格元素
nRow2, nCol2	要交换的第二个表格元素

表 4-63：***setUndoTextToText / setUndoValueToValue / undoText / undoValue***

setUndoTextToText: 在指定的表格元素中设置 undo 文本。 ***setUndoValueToValue:*** 在指定的表格元素中设置 undo 值。 ***undoText:*** 提供当前 undo 文本。 ***undoValue:*** 提供当前 undo 值。
void SlGrGrid::setUndoTextToText(int nRow, int nCol); **void SlGrGrid::setUndoValueToValue(int nRow, int nCol);** **QString SlGrGrid::undoText(int nRow, int nCol) const;** **QVariant SlGrGrid::undoValue(int nRow, int nCol) const;**

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标

表 4-64：***updateCell***

updateCell: 更新指定的表格元素。
void SlGrGrid::updateCell(int nRow, int nCol);

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标

表 4-65：***updateContents***

updateContents: 更新可见区域。
void updateContents();

函数 – 单元格的可见性

表 4-66：***setCurrentCell / currentRow / currentColumn***

setCurrentCell: 将光标置于指定的表格元素中。 ***currentRow:*** 提供光标当前所在的行。 ***currentColumn:*** 提供光标当前所在的列。
void SlGrGrid::setCurrentCell(int nRow, int nCol, bool bUpdateRows = false); **int SlGrGrid::currentRow() const;** **int SlGrGrid::currentColumn() const;**

参数	含义
nRow	行
nCol	列
bUpdateRows	无含义（不传送任何内容）

表 4-67：***ensureCellVisible***

滚动表格，直至指定的表格元素可显示。
void SlGrGrid::ensureCellVisible(int nRow, int nCol);

参数	含义
nRow	要显示的单元格的行下标
nCol	要显示的单元格的列下标

表 4-68：***topRowVisible / bottomRowVisible / rightColVisible / leftColVisible***

topRowVisible / bottomRowVisible: 提供表格可显示区域的最上面或最下面一行。 ***rightColVisible / leftColVisible:*** 提供表格可见区域的最右侧或最左侧一列。
int SlGrGrid::topRowVisible(bool bComplete = true) const; **int SlGrGrid::bottomRowVisible(bool bComplete = true) const;** **int SlGrGrid::rightColVisible(bool bComplete = true) const;** **int SlGrGrid::leftColVisible(bool bComplete = true) const;**

参数	含义
bComplete	TRUE: 只针对可完整显示的行或列。
返回值	行下标或列下标

表 4-69：***hideRow / showRow / isRowHidden***
hideColumn / showColumn / isColumnHidden

hideRow: 隐藏一行。 ***showRow:*** 使被隐藏的行重新显示。 ***isRowHidden:*** 告知，一行是隐藏还是可见。 ***hideColumn:*** 隐藏一列。 ***showColumn:*** 使被隐藏的列重新显示。 ***isColumnHidden:*** 告知，一列是隐藏还是可见。
void SlGrGrid::hideRow(int nRow); **void SlGrGrid::showRow(int nRow);** **bool SlGrGrid::isRowHidden(int nRow) const;** **void SlGrGrid::hideColumn(int nCol);** **void SlGrGrid::showColumn(int nCol);** **bool SlGrGrid::isColumnHidden(int nCol) const;**

参数	含义
nRow	要隐藏或重新显示的行。
nCol	要隐藏或重新显示的列。
返回值	TRUE: 行或列被隐藏。 FALSE: 行或列可见。

表 4-70：***visibleRows***

指定表格可见区域的行数。
int SlGrGrid::visibleRows() const;

参数	含义
返回值	可见行数

函数 – 行和列的写保护

表 4-71：***setRowReadOnly / isRowReadOnly***
setColumnReadOnly / isColumnReadOnly

setRowReadOnly: 设置或取消行的写保护。 ***isRowReadOnly:*** 告知，一行是否被写保护。 ***setColumnReadOnly:*** 设置或取消列的写保护。 ***isColumnReadOnly:*** 告知，一列是否被写保护。 （另见属性“readOnly”）
void SlGrGrid::setRowReadOnly(int nRow, bool bReadO); **bool SlGrGrid::isRowReadOnly(int nRow) const;** **void SlGrGrid::setColumnReadOnly(int nCol, bool bReadO);** **bool SlGrGrid::isColumnReadOnly(int nCol) const;**

参数	含义
nRow	要设置或取消写保护的行。
nCol	要设置或取消写保护的列。
bReadO	TRUE: 为行或列设置写保护。 FALSE: 为行或列取消写保护。
返回值	TRUE: 行或列被写保护。 FALSE: 行或列可编辑。

函数 - 选中单元格

表 4-72：***addSelection***

向表格添加一个选择并为此分配一个 ID。
int SlGrGrid::addSelection(const SlGrGridSelection& rSel);

参数	含义
rSel	要添加的选择
返回值	新选择的 ID 或当选择无效时返回-1。

表 4-73：***selection***

返回 ID 所对应的选择。如果 ID 不在有效范围内，则返回无效选择。
SlGrGridSelection SlGrGrid::selection(int nNum) const;

参数	含义
nNum	选择对应的 ID

表 4-74：***isSelectedArea***

告知，选中是否存在。
bool SlGrGrid::isSelectedArea() const;

参数	含义
返回值	TRUE: 选中了一个区域。 FALSE: 未在表格中进行选择。 （选中方式为 SINGLE_ROW 时，始终返回 TRUE；为 NO_SELECTION 时，始终返回 FALSE）

表 4-75：***selectCells / selectRow***

selectCells: 选中指定区域。 ***selectRow:*** 选中指定行。
void SlGrGrid::selectCells(int nStartRow, int nEndRow); **void SlGrGrid::selectRow(int nRow);**
参数	含义
nStartRow	选中区域开始的行。
nEndRow	选中区域结束的行。
nRow	要选中的行

表 4-76：***removeSelection***

取消表格中已有的选中。此时可以指定 SlGrGridSelection 对象或选择对应的 ID。
void SlGrGrid::removeSelection(const SlGrGridSelection& rSel); **void SlGrGrid::removeSelection(int nNum);**
参数	含义
rSel	要取消的选中
nNum	要取消的选中所对应的 ID

表 4-77：***clearSelection***

取消表格中所有已有的选中。
void SlGrGrid::clearSelection(bool bRepaint = true);
参数	含义
bRepaint	无含义（不传送任何内容）

表 4-78：***isSelected / isRowSelected / isColumnSelected***

isSelected: 告知，指定单元格是否被选中。 ***isRowSelected:*** 告知，指定行是否被选中。 ***isColumnSelected:*** 告知，指定列是否被选中。
bool SlGrGrid::isSelected(int nRow, int nCol) const; **bool SlGrGrid::isRowSelected(int nRow) const;** **bool SlGrGrid::isColumnSelected(int nCol, bool bFull = false) const;**
参数	含义
nRow	行
nCol	列
bFull	TRUE: 如果选中列的所有单元格，返回值只显示 TRUE。 FALSE: 如果选中列的至少一个单元格，返回值显示 TRUE。
返回值	TRUE: 单元格或行或列被选中。 FALSE: 单元格或行或列未被选中。

函数 – 颜色管理

支持以下 ColorRole：

表 4-79： 支持的 ColorRole

ColorRole	含义/颜色，用于描绘…
QPalette::Base	…未获得焦点的未选中的单元格
QPalette::Highlight	…获得了焦点的未选中的单元格
QPalette::Link	…未获得焦点的被选中的单元格
QPalette::Button	…获得了焦点的被选中的单元格
QPalette::Text	…未选中单元格的文本
QPalette::Light	…被选中单元格的文本
QPalette::Mid	…不可编辑的单元格的外框
QPalette::Shadow	…栅格
QPalette::Dark	…带光标的单元格，但表格未获得焦点

表 4-80：***setCellColorPalette / getCellColorPalette / removeCellColorPalette***

setCellColorPalette: 为指定单元格设置自己的（专用的）调色板。 ***getCellColorPalette:*** 提供指定单元格的调色板。 ***removeCellColorPalette:*** 取消指定单元格的专用调色板。此后表格的通用调色板重新适用于该单元格。 提示!!! 在颜色管理中这些函数的优先级最高。
void SlGrGrid::setCellColorPalette(int nRow, int nCol, QPalette palette); **QPalette SlGrGrid::getCellColorPalette(int nRow, int nCol, bool& rbPal) const;** **void SlGrGrid::removeCellColorPalette(int nRow, int nCol);**

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
rbPal	TRUE: 该单元格的专用调色板。 FALSE: 整个表格的通用调色板。
palette	调色板

表 4-81：***setRowColorPalette / getRowColorPalette / removeRowColorPalette***

setRowColorPalette: 为指定行中的所有单元格设置自己的（专用的）调色板。 ***getRowColorPalette:*** 提供指定行的调色板。 ***removeRowColorPalette:*** 取消指定行的专用调色板。此后表格的通用调色板重新适用于该行。 提示!!! 在颜色管理中这些函数具有第二高的优先级。
void SlGrGrid::setRowColorPalette(int nRow, QPalette palette); **QPalette SlGrGrid::getRowColorPalette(int nRow, bool& rbPal) const;** **void SlGrGrid::removeRowColorPalette(int nRow);**

参数	含义
nRow	行
rbPal	TRUE: 该行的专用调色板。 FALSE: 整个表格的通用调色板。
palette	调色板

表 4-82：***setColumnColorPalette / getColumnColorPalette***
removeColumnColorPalette

setColumnColorPalette: 为指定列中的所有单元格设置自己的（专用的）调色板。 ***getColumnColorPalette:*** 提供指定列的调色板。 ***removeColumnColorPalette:*** 取消指定列的专用调色板。此后表格的通用调色板重新适用于该列。 提示!!! 在颜色管理中这些函数的优先级最低。
void SlGrGrid::setColumnColorPalette(int nCol, QPalette palette); **QPalette SlGrGrid::getColumnColorPalette(int nCol, bool& rbPal) const;** **void SlGrGrid::removeColumnColorPalette(int nCol);**

参数	含义
nCol	列
rbPal	TRUE: 该列的专用调色板。 FALSE: 整个表格的通用调色板。
palette	调色板

表 4-83：***getColorPalette***

提供调色板以及指定单元格的调色板的源。
QPalette SlGrGrid::getColorPalette(int nRow, int nCol, int& rnType) const;

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
rnType	0: 表格的通用调色板 1: 该单元格的专用调色板 2: 该行的专用调色板 3: 该列的专用调色板
返回值	单元格的调色板

表 4-84：***isSpecialPalette***

告知，是否为指定单元格设置了专用调色板。
bool SlGrGrid::isSpecialPalette(int nRow, int nCol) const;

参数	含义
nRow	单元格的行下标
nCol	单元格的列下标
返回值	TRUE: 为单元格设置了专用调色板。 FALSE: 没有为单元格设置专用调色板。

函数 – 字体管理

表 4-85：***setFont***

设置整个表格的字体并调整行高。 前提应已设置“setHmiFonts(false)”。
void SlGrGrid::setFont(const QFont& rFont);

参数	含义
rFont	新字体

表 4-86：***setHmiFontSize***

setHmiFontSize: 设置整个表格以及行标题或列标题的 HMI 字体的像素大小。 前提应已设置“setHmiFonts(true)”。 提示!!! 函数“setRowFontSize”和”setColumnFontSize”具有较高的优先级，即使用这些函数设置的像素大小不能使用“setHmiFontSize”修改。
void SlGrGrid::setHmiFontSize(int nHmiFontSize);

参数	含义
nHmiFontSize	要设置的像素大小

表 4-87：***setRowFontSize / getRowFontSize***
setColumnFontSize / getColumnFontSize

setRowFontSize / setColumnFontSize: 设置各指定行或列的 HMI 字体的像素大小。 前提应已设置“setHmiFonts(true)”。 ***getRowFontSize / getColumnFontSize***:提供之前为各行或列设置的像素大小。
void SlGrGrid::setRowFontSize(int nRow, int nFontSize); **int SlGrGrid::getRowFontSize(int nRow) const;** **void SlGrGrid::setColumnFontSize(int nCol, int nFontSize);** **int SlGrGrid::getColumnFontSize(int nCol) const;**

参数	含义
nFontSize	要设置的像素大小
nRow	行
nCol	列

函数 – 表格标题

表 4-88：***horizontalHeader / verticalHeader***

horizontalHeader: 提供上方的水平标题行。 ***verticalHeader:*** 提供左侧的垂直标题列。
SlGrHeader* SlGrGrid::horizontalHeader() const; **SlGrHeader* SlGrGrid::verticalHeader() const;**

参数	含义
返回值	作为 SlGrHeader 指针的相应的表格标题。

表 4-89：***setRowLabels / setColumnLabels***

setRowLabels: 设置垂直标题列的标题内容。 ***setColumnLabels:*** 设置水平标题行的标题内容。
void SlGrGrid::setRowLabels(const QStringList& rLabels) const; **void SlGrGrid::setColumnLabels(const QStringList& rLabels) const;**

参数	含义
rLabels	标题文本的列表

函数 – 调整表格

表 4-90：***setColumnWidth / setRowHeight / columnWidth / rowHeight***

setColumnWidth: 设置指定列的宽度。
setRowHeight: 设置指定行的高度。
columnWidth: 提供指定列的宽度。
rowHeight: 提供指定行的高度。

```
void    SlGrGrid::setColumnWidth(int nCol, int nPix);
void    SlGrGrid::setRowHeight(int nRow, int nPix, bool bPermanent = true);
int     SlGrGrid::columnWidth(int nCol) const;
int     SlGrGrid::rowHeight(int nRow) const;
```

参数	含义
nRow	行
nCol	列
nPix	要设置的宽度或高度
bPermanent	TRUE: 更改字体不调整行高。 FALSE: 更改字体会调整行高。

表 4-91：***adjustColumn / adjustRow***

adjustColumn: 更改指定列的宽度，使该列中最长的条目也能完全显示。

adjustRow: 更改指定行的高度，使该行中最高的条目也能完全显示。

```
void SlGrGrid::adjustColumn(int nCol);
void SlGrGrid::adjustRow(int nRow);
```

参数	含义
nRow	行
nCol	列

表 4-92：***setColumnStretchable / setRowStretchable***
isColumnStretchable / isRowStretchable

setColumnStretchable: 确定指定列的宽度是否可变，即列宽自动调整，以尽量使所有列无需滚动就都可见。

setRowStretchable: 确定指定行的高度是否可变，即行高自动调整，以尽量使所有行无需滚动就都可见。

isColumnStretchable: 告知，指定列的宽度是否可变。

isRowStretchable: 告知，指定行的高度是否可变。

```
void SlGrGrid::setColumnStretchable(int nCol, bool bStretch);
void SlGrGrid::setRowStretchable(int nRow, bool bStretch);
bool SlGrGrid::isColumnStretchable(int nCol) const;
bool SlGrGrid::isRowStretchable(int nRow) const;
```

参数	含义
nRow	行
nCol	列
bStretch	TRUE: 行宽或列高可变。 FALSE: 行宽或列高不可变。

表 4-93：***setLeftMargin / setTopMargin***

setLeftMargin: 设置左侧垂直标题列宽（verticalHeader）。 ***setTopMargin:***设置上部水平标题行高（horizontalHeader）。
void SlGrGrid::setLeftMargin(int nPix); **void SlGrGrid::setTopMargin(int nPix);**

参数	含义
nPix	宽/高（单位：像素）

表 4-94：***setColumnAlignment***

设置指定列的所有单元格中文本和 pixmap 的对齐方式。
void SlGrGrid::setColumnAlignment(int nCol, Qt::Alignment nTextAlign = Qt::AlignJustify, Qt::Alignment nPixmapAlign = Qt::AlignHCenter);

参数	含义
nCol	列
nTextAlign	文本的对齐方式。允许的方式有： Qt::AlignLeft, Qt::AlignRight, Qt::AlignHCenter,Qt::AlignJustify 缺省设置为 Qt::AlignJustify。此时 SlGrGrid 自己来控制对齐方式，即普通文本左对齐，含数字的文本右对齐。
nPixmapAlign	pixmap 的对齐方式。

表 4-95：***sortColumn***

对指定列的单元格进行排序。 提示!!! 该函数的调用不影响表头中指定排序顺序的箭头。
void SlGrGrid::sortColumn(int nCol, bool bAscending = true, bool bWholeRows = false, bool bSwapHeader = false);

参数	含义
nCol	列
bAscending	TRUE: 升序 FALSE: 降序
bWholeRows	TRUE: 按需要的排序顺序对整行内容进行移动。 FALSE: 只对指定列的单元格进行排序（对其他单元格无效）。
bSwapHeader	TRUE: 按需要的排序顺序对行标题进行移动。 FALSE: 标题保持不变。

表 4-96：***setFrame***

激活或取消表格边界。
void SlGrGrid::setFrame(bool bSetFrame);

参数	含义
bSetFrame	TRUE: 表格带有边界。 FALSE: 表格不带边界。

表 4-97：***setCellNoFocus***

确定哪些单元格可获得焦点。如果取消了一个单元格的焦点，则无法再使用鼠标或光标键选中该单元格。焦点会被置于下一个单元格中。默认所有的单元格都可选中。该函数对空单元格无效。	
void SlGrGrid::setCellNoFocus(int nRow, int nCol, bool bLock);	
参数	含义
int nRow	单元格的行下标
int nCol	单元格的列下标
bool bLock	TRUE: 单元格不获得焦点 FALSE: 单元格可获得焦点（缺省设置）

类 – SlGrHeader

标题行或标题列可通过 SlGrHeader 类进行调整。 相应标题区域的指针通过函数“horizontalHeader”或“verticalHeader”获取。各个行标题或列标题称为段(section)。

表 4-98：SlGrHeader 的属性

属性	读取/ 写入	描述
bool tracking	tracking setTracking	TRUE: 当鼠标移动时（段的大小也会变化），会连续发送“sizeChange()”信号。 FALSE: 当鼠标键被松开并且段的大小调整也结束时，才会最终发送一次信号“sizeChange()”。
int count	count	提供段数。

表 4-99：SlGrHeader 的信号

信号	参数	描述
clicked()	当在一个段上单击（按下又松开）鼠标左键时，触发该信号。	
	int nSection	段的下标
pressed()	当在一个段上按下鼠标左键时，触发该信号。	
	int nSection	段的下标
released()	当在一个段上松开鼠标左键时，触发该信号。	
	int nSection	段的下标
sizeChange()	当段的大小被改变时，触发该信号。	
	int nSection	段的下标
	int nOldSize	老的段大小
	int nNewSize	新的段大小
indexChange()	当一个段被移动到其他位置时，触发该信号。 （见“rowMovingEnabled”和“columnMovingEnabled”）	
	Int nSection	段的下标
	int nFromIndex	老位置的下标
	int nToIndex	新位置的下标

表 4-100：***setLabel / label / iconSet***

setLabel: 设置段的文本或图标。 ***label:*** 提供段的文本。如果该段不存在，则返回 QString::null。 ***iconSet:*** 提供段的图标。如果该段不存在，则返回 0。
void SlGrHeader::setLabel(int nSection, const QString& rString, int nSize = -1); **void SlGrHeader::setLabel(int nSection, const QIcon& rIconSet,** **const QString& rString, int nSize = -1);** **QString SlGrHeader::label(int nSection) const;** **QIcon* SlGrHeader::iconSet(int nSection) const;**

参数	含义
nSection	段的下标
rString	待设置的文本
rIconSet	待设置的图标
nSize	待设置的段的宽度。如在此处给定一个负值，则宽度不变。

表 4-101：***setSectionAlignment / getSectionAlignment***

setSectionAlignment: 设置段的水平对齐方式。 ***getSectionAlignment:*** 提供段的水平对齐方式。如果该段不存在，则返回-1。
void SlGrHeader::setSectionAlignment(int nSection, **Qt::Alignment nAlign,** **Qt::Alignment nAlignIcon = Qt::AlignLeft);** **Qt::Alignment SlGrHeader::getSectionAlignment(int nSection) const;**

参数	含义
nSection	段的下标
nAlign	文本的水平对齐方式。允许的方式有： Qt::AlignLeft, Qt::AlignRight, Qt::AlignHCenter 垂直方向上文本始终居中对齐。
nAlignIcon	无含义（不传送任何内容）

表 4-102：***setClickEnabled / isClickEnabled***

setClickEnabled: 确定，在点击指定段时是否触发“clicked()”信号（即该段是“可点击的”）。如传递-1，则适用于所有存在的段以及之后添加的所有段。 ***isClickEnabled:*** 告知，该段是否可点击。
void SlGrHeader::setClickEnabled(bool bEnable, int nSection = -1); **bool SlGrHeader::isClickEnabled(int nSection = -1) const;**

参数	含义
nSection	段的下标 （-1 时，该修改影响所有段）
bEnable	TRUE: “clicked()”信号被触发。 FALSE: 对段的点击被忽略。

表 4-103：***setResizeEnabled / isResizeEnabled***

setResizeEnabled: 确定，是否可通过鼠标修改指定段的宽度/高度。如传递-1，则适用于所有存在的段以及之后添加的所有段。 ***isResizeEnabled:*** 告知，是否可使用鼠标修改段的宽度/高度。	
void SlGrHeader::setResizeEnabled(bool bEnable, int nSection = -1); **bool SlGrHeader::isResizeEnabled(int nSection = -1) const;**	
参数	含义
nSection	段的下标 （-1 时，该修改影响所有段）
bEnable	TRUE: 段的大小可以更改。 FALSE: 段的大小不可更改。

类 – SlGrGridItem

SlGrGridItem 类代表一个 SlGrGrid 单元格。其为一个标准数据区，可包含文本、数据和/或 pixmap。 通过函数“setItem”将“SlGrGridItem”添加到“SlGrGrid”中。

基本类 SlGrGridItem 还有以下这些特殊的派生类（见后面的介绍）：

- SlGrCheckGridItem ➔ 复选框元素
- SlGrComboGridItem➔ 下拉列表框元素
- SlGrToggleGridItem➔ 切换框元素

表 4-104：***SlGrGridItem 的构造函数***

SlGrGridItem(SlGrGrid* pTable, EditTypeEnum eType); **SlGrGridItem(SlGrGrid* pTable, EditTypeEnum eType, const QVariant& rValue);** **SlGrGridItem(SlGrGrid* pTable, EditTypeEnum eType, const QVariant& rValue, const QPixmap& rPix);**	
参数	含义
pTable	要向其中分配新单元格的表格的指针。
eType	单元格的编辑模式： NEVER: 单元格无法编辑。 ON_TYPING: 单元格可以编辑。
rValue	要显示的数据或文本
rPix	要显示的 pixmap

此时应注意：

1) 删除 SlGrGridItem 会导致其从所属的 SlGrGrid 中被删除。因此 SlGrGridItem 应始终创建在堆存储器上。
2) 一个 SlGrGridItem 只能分配给一个 SlGrGrid。

表 4-105：***undoValue***

undoValue: 提供 undo 值。
QVariant& SlGrGridItem::undoValue() const;

表 4-106：***displayText***

displayText: 提供显示值。	
void SlGrGridItem:: displayText();	
参数	含义

表 4-107：***setValue / value / setText / text***

setValue: 设置一个值。 ***setText:*** 设置一个文本。 ***value:*** 提供所设置的值。 ***text:*** 提供作为文本的值，或在文本不存在时返回 QString::null。	
void SlGrGridItem::setValue(const QVariant& rValue, bool bSetUndo = true); void SlGrGridItem::setText(const QString& rString, bool bSetUndo = true); QVariant& SlGrGridItem::value() const; QString SlGrGridItem::text() const;	
参数	含义
rValue	待设置的值
rString	待设置的文本
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-108：***setTextAlignment***

setTextAlignment: 设置文本的对齐方式。	
void SlGrGridItem::setTextAlignment(Qt::Alignment nAlign);	
参数	含义
nAlign	文本的对齐方式。允许的方式有： Qt::AlignLeft, Qt::AlignRight, Qt::AlignHCenter,Qt::AlignJustify 缺省设置为 Qt::AlignJustify。此时 SlGrGrid 自己来控制对齐方式，即普通文本左对齐，含数字的文本右对齐。

表 4-109：***setPixmap / pixmap***

setPixmap: 设置一个 pixmap。 ***pixmap:*** 提供所设置的 pixmap，或当不存在时返回零 pixmap。	
void SlGrGridItem::setPixmap(const QPixmap& rPix, Qt::Alignment nAlign = Qt::AlignHCenter); QPixmap SlGrGridItem::pixmap() const;	
参数	含义
rPix	待设置的 pixmap
nAlign	pixmap 的对齐。全部都可能为 Qt 对齐方式，除了 Qt::AlignJustify 和 Qt::AlignAbsolute。

表 4-110：***setPixmapAlignment***

setPixmapAlignment: 设置 pixmap 的对齐方式。	
void SlGrGridItem::setPixmapAlignment(Qt::Alignment nAlign);	
参数	含义
nAlign	pixmap 的对齐。全部都可能为 Qt 对齐方式，除了 Qt::AlignJustify 和 Qt::AlignAbsolute。

表 4-111：***setWordWrap / wordWrap***

setWordWrap: 激活/取消文本换行。 ***wordWrap:*** 告知，换行是否激活。	
void SlGrGridItem::setWordWrap(bool bWrap); bool SlGrGridItem::wordWrap() const;	
参数	含义
bWrap	TRUE: 超出单元格范围的文本进行换行。 （必须设置属性“multicellularAllowed”）。 FALSE:文本不进行换行。

表 4-112：***setEditType / editType***

setEditType:设置单元格的编辑模式。 ***editType:*** 提供单元格的编辑模式。	
void	**SlGrGridItem::setEditType(const EditTypeEnum nType, bool bWithUpdate = true);** **EditTypeEnum SlGrGridItem::editType() const;**
参数	含义
nType	单元格的编辑模式： NEVER: 单元格无法编辑。 ON_TYPING: 单元格可以编辑。

表 4-113：***table***

table: 返回单元格所属表格的指针。
SlGrGrid* SlGrGridItem::table() const;

表 4-114：***sizeHint***

sizeHint: 提供单元格的大小，以使其全部内容都可见。
QSize SlGrGridItem::sizeHint();

表 4-115：***setSpan / rowSpan / colSpan***

setSpan: 更改单元格的大小，以便占据多列或多行。此时左上方的单元格为原始单元格。前提是单元格已通过“setItem”分配给了一个表格。函数此时会检查，新的单元格大小是否超出了所属表格的大小。如果超出，函数将不起作用。不支持对与这样的单元格相交的列或行进行交换、添加或删除的函数。 ***rowSpan:*** 提供单元格占据的行数。 ***colSpan:*** 提供单元格占据的列数。	
void **int** **int**	**SlGrGridItem::setSpan(int nRows, int nCols);** **SlGrGridItem::rowSpan() const;** **SlGrGridItem::colSpan() const;**
参数	含义
nRows	单元格要占据的行数。
nCols	单元格要占据的列数。

表 4-116：***setEnabled / isEnabled***

setEnabled: 激活/取消用户与单元格的即时互动。 ***isEnabled:***告知，用户即时互动是否可用。	
void SlGrGridItem::setEnabled(bool bEnabled); **bool SlGrGridItem::isEnabled() const;**	
参数	含义
bEnabled	TRUE: 用户即时互动可用 FASLE: 无用户即时互动。

表 4-117：***setInputMode / inputMode***

setInputMode: 设置单元格的输入模式（缺省设置：ANY_MODE）。 ***inputMode:*** 提供单元格的输入模式。	
void SlGrGridItem::setInputMode(SlGfwFunctions::SlGfwDisplayModeEnum inputMode); **SlGfwFunctions::SlGfwDisplayModeEnum SlGrGridItem::inputMode() const;**	
参数	含义
inputMode	参见章节“GUI 组件/HMI 小部件/SlGfwLineEdit / Inputmode”中的表“Inputmode”

表 4-118：***setDecimals / decimals***

setDecimals: 设置小数点后的位数（缺省设置：3）。 这只在输入模式(UN)SIGNED_DOUBLE_MODE 和 DOUBLE_EXPONENT_MODE 下有效。 ***decimals:*** 提供单元格的输入模式。
void SlGrGridItem::setDecimals(int nDecimals); **int SlGrGridItem::decimals() const;**

参数	含义
nDecimals	小数点后的位数

表 4-119：***setLockFocus / isLockFocus***

setLockFocus: 激活/取消单元格可获得焦点的属性。 ***isLockFocus:*** 告知，单元格是否可获得焦点。
void SlGrGridItem::setLockFocus(bool bLock); **bool SlGrGridItem::isLockFocus() const;**

参数	含义
bLock	TRUE: 单元格无法获得焦点。 FALSE: 单元格可获得焦点。

表 4-120：***setMinimumValue / minimumValue***
setMaximumValue / maximumValue

setMinimumValue: 设置单元格的下限值。 ***setMaximumValue:*** 设置单元格的上限值。 这只在输入模式(UN)SIGNED_INTEGER_MODE、(UN)SIGNED_DOUBLE_MODE、(UN)SIGNED_EXPONENT_MODE 下有效。 ***minimumValue:*** 提供单元格的下限值。 ***maximumValue:*** 提供单元格的上限值。
void SlGrGridItem::setMinimumValue(double dMinimumValue); **double SlGrGridItem::minimumValue(void) const;** **void SlGrGridItem::setMaximumValue(double dMaximumValue);** **double SlGrGridItem::maximumValue(void) const;**

参数	含义
dMinimumValue	下限值
dMaximumValue	上限值

表 4-121：***setEnableInternalCalculator / enableInternalCalculator***

setEnableInternalCalculator: 激活/取消内部计算器。以此可直接进行计算（例如 5+5），而无需将计算器显示出来（见“setCalculatorEnabled”）。 ***enableInternalCalculator:*** 告知，内部计算器是否激活。
void SlGrGridItem::setEnableInternalCalculator(bool bEnable); **bool SlGrGridItem::enableInternalCalculator(void) const;**

参数	含义
bEnable	TRUE: 内部计算器被激活。 FALSE: 内部计算器被取消。

表 4-122：***setCalculatorEnabled / isCalculatorEnabled***

setCalculatorEnabled: 激活/取消计算器。输入等号“=”即可显示计算器。 通过设置某些“输入模式”可自动激活计算器（见“GUI 组件/HMI 小部件/SlGfwLineEdit/计算器功能”）。 此外还要将计算器屏幕加入到对话框定义中： `<IMPORT file="L:/gui/dialogs/common/slgfwcommon incl.xml" screen="SlGfwCalculatorScreen" />` ***isCalculatorEnabled:*** 告知，计算器是否激活。
void SlGrGridItem::setCalculatorEnabled(bool bCalculatorEnabled); **bool SlGrGridItem::isCalculatorEnabled(void) const;**

参数	含义
bCalculatorEnabled	TRUE: 计算器被激活。 FALSE: 计算器被取消。

表 4-123：***setRegExpInputValidator / regExpInputValidator***

setRegExpInputValidator: 设置正则表达式输入验证。 ***regExpInputValidator:*** 提供正则表达式输入验证。
void SlGrGridItem::setRegExpInputValidator(const QString& rszRegExpInputValidator); **const QString& SlGrGridItem::regExpInputValidator() const;**

参数	含义
rszRegExpInputValidator	正则表达式字符串

类 – SlGrCheckGridItem

SlGrCheckGridItem 类是 SlGrGridItem 类的一个派生类，代表一个 SlGrGrid 单元格。它是一个复选框元素。 通过函数“setItem”将“SlGrCheckGridItem”添加到“SlGrGrid”中。

表 4-124：***SlGrCheckGridItem 的构造函数***

SlGrCheckGridItem(SlGrGrid* pTable, const QString& rString);	
参数	含义
pTable	要向其中分配新单元格的表格的指针。
rString	待显示的文本

除 SlGrGridItem 的注意事项外，此时还应注意：

1) 在 SlGrCheckGridItem 中不能使用 pixmap。

表 4-125：***setChecked / isChecked***

setChecked: 激活/取消单元格的复选框。 ***isChecked:*** 告知，是否选中了单元格的复选框。
void SlGrCheckGridItem::setChecked(bool bCheck, bool bWithUpdate = true); **bool SlGrCheckGridItem::isChecked() const;**

参数	含义
bCheck	TRUE: 单元格的复选框显示为已选中。 FALSE: 单元格的复选框显示为未选中。

表 4-126：***setIconAlignment / iconAlignment***

setIconAlignment: 设置复选框的对齐方式。 ***iconAlignment:*** 提供复选框的对齐方式。	
void SlGrCheckGridItem::setIconAlignment(Qt::Alignment nAlign); **Qt::Alignment SlGrCheckGridItem::iconAlignment() const;**	
参数	含义
nAlign	复选框的对齐方式。允许的方式有： Qt::AlignLeft, Qt::AlignRight

表 4-127：***setText***

setText: 设置复选框旁标签的文本。	
void SlGrCheckGridItem::setText(const QString& rString, bool bSetUndo = true);	
参数	含义
rString	待设置的文本
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

类 – SlGrComboGridItem

SlGrComboGridItem 类是 SlGrGridItem 类的一个派生类，代表一个 SlGrGrid 单元格。它是一个下拉列表框元素。 通过函数“setItem”将“SlGrComboGridItem”添加到“SlGrGrid”中。

表 4-128：***SlGrComboGridItem 的构造函数***

SlGrComboGridItem(SlGrGrid* pTable, const QStringList& rList, bool bEditable = false);	
参数	含义
pTable	要向其中分配新单元格的表格的指针。
rList	要填入到单元格中的文本列表。
bEditable	TRUE: 单元格可以编辑。 FALSE: 单元格无法编辑。

除 SlGrGridItem 的注意事项外，此时还应注意：

1) 在 SlGrComboGridItem 中只能显示文本。

表 4-129：***setCurrentItem / currentItem / currentText***

setCurrentItem: 设置所选择的列表元素，通过下标或文本。如不存在文本，则不产生影响。 ***currentItem:***提供所选列表元素的下标。 ***currentText:***提供所选列表元素的文本。	
void SlGrComboGridItem::setCurrentItem(int nIndex, bool bWithUpdate = true, bool bSetUndo = true); **void SlGrComboGridItem::setCurrentItem(const QString& rString, bool bWithUpdate = true, bool bSetUndo = true);** **int SlGrComboGridItem::currentItem() const;** **QString SlGrComboGridItem::currentItem() const;**	
参数	含义
nIndex	列表元素的下标
rString	列表元素的文本
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-130：***count***

count:提供列表元素的数量。
int SlGrComboGridItem::count() const;

表 4-131：***text***

text: 提供通过下标选择的列表元素的文本。	
QString SlGrComboGridItem::text(int nIndex) const;	
参数	含义
nIndex	列表元素的下标

表 4-132：***setEditable / isEditable***

setEditable: 激活/取消可编辑性。如果单元格可编辑，可在单元格中输入自己的文本，再通过按下 return 键将文本添加到列表元素中。 ***isEditable:*** 告知，单元格是否可编辑。	
void SlGrComboGridItem::setEditable(bool bEdit); **bool SlGrComboGridItem::isEditable() const;**	
参数	含义
bEdit	TRUE: 单元格可以编辑。 FALSE: 单元格无法编辑。

表 4-133：***setStringList***

setStringList: 将传递的 QStringList 设为列表元素。已存在的列表元素会被删除。	
void SlGrComboGridItem::setStringList(const QStringList& rList, bool bWithUpdate = true, bool bSetUndo = true);	
参数	含义
rList	要填入到单元格中的文本列表。
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-134：***clearItems***

clearItems:清除所有列表元素。
void SlGrComboGridItem::clearItems(bool bWithUpdate = true);

类 – SlGrToggleGridItem

SlGrToggleGridItem 类是 SlGrGridItem 类的一个派生类，代表一个 SlGrGrid 单元格。它是一个切换框元素（不可编辑的下拉列表框）。 通过函数“setItem”将“SlGrToggleGridItem”添加到“SlGrGrid”中。

表 4-135：***SlGrToggleGridItem 的构造函数***

SlGrToggleGridItem(SlGrGrid* pTable, const QStringList& rList);	
参数	含义
pTable	要向其中分配新单元格的表格的指针。
rList	要填入到单元格中的文本列表。

除 SlGrGridItem 的注意事项外，此时还应注意：

1) 在 SlGrToggleGridItem 中只能显示文本。

表 4-136：***setCurrentItem / currentItem / currentText***

setCurrentItem: 设置所选择的列表元素，通过下标或文本。如不存在文本，则不产生影响。 ***currentItem:***提供所选列表元素的下标。 ***currentText:***提供所选列表元素的文本。
void SlGrToggleGridItem::setCurrentItem(int nIndex, bool bWithUpdate = true, bool bSetUndo = true); **void SlGrToggleGridItem::setCurrentItem(const QString& rString, bool bWithUpdate = true, bool bSetUndo = true);** **int SlGrToggleGridItem::currentItem() const;** **QString SlGrToggleGridItem::currentText() const;**

参数	含义
nIndex	列表元素的下标
rString	列表元素的文本
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-137：***count***

count:提供列表元素的数量。
int SlGrToggleGridItem::count() const;

表 4-138：***text***

text: 提供通过下标选择的列表元素的文本。
QString SlGrToggleGridItem::text(int nIndex) const;

参数	含义
nIndex	列表元素的下标

表 4-139：***setStringList***

setStringList: 将传递的 QStringList 设为列表元素。已存在的列表元素会被删除。
void SlGrToggleGridItem::setStringList(const QStringList& rList, bool bWithUpdate = true, bool bSetUndo = true);

参数	含义
rList	要填入到单元格中的文本列表。
bSetUndo	TRUE: undo 值被覆盖。 FALSE: undo 值被保留。

表 4-140：***clearItems***

clearItems:清除所有列表元素。
void SlGrToggleGridItem::clearItems(bool bWithUpdate = true);

类 – SlGrGridSelection

在选中表格的单元格时需要使用 SlGrGridSelection 类（另见“GUI 组件/HMI 小部件/SlGrGrid/函数 - 选中单元格”）。

此处的选中区域为带有锚点单元格的矩形单元格组，所谓的锚点单元格为第一个选中的单元格。

注

凭目测只能选中整行。

表 4-141：***SlGrGridSelection 的构造函数***

SlGrGridSelection(); ➔ 产生空白无效的选择。如要激活该选择，必须执行初始化(init)并扩展(expandTo)。 **SlGrGridSelection(int nStartRow, int nStartCol, int nEndRow, int nEndCol);** ➔ 产生有效的选择。

参数	含义
nStartRow	选中区域开始的行。
nStartCol	选中区域开始的列。
nEndRow	选中区域结束的行。
nEndCol	选中区域结束的列。

表 4-142：***init***

init: 设置选中区域的锚点单元格。进行该调用后该选中区域将含有该单元格，但选中不生效。 欲将其激活，要使用函数“expandTo”。
void SlGrGridSelection::init(int nRow, int nCol);

参数	含义
nRow	锚点单元格的行下标。
nCol	锚点单元格的列下标。

表 4-143：***expandTo***

expandTo: 生成矩形选中区域，从锚点单元格开始，到此处传递的行下标/列下标为止。选中在该调用后生效。 之前未调用“init”，则不产生影响。
void SlGrGridSelection::expandTo(int nRow, int nCol);

参数	含义
nRow	选中区域要扩展到的行。
nCol	选中区域要扩展到的列。

表 4-144：***topRow / bottomRow / leftCol / rightCol***

topRow: 提供选中区域的最上面一行。 ***bottomRow:*** 提供选中区域的最下面一行。 ***leftCol:*** 提供选中区域的最左侧一列。 ***rightCol:*** 提供选中区域的最右侧一列。
int SlGrGridSelection::topRow() const; **int SlGrGridSelection::bottomRow() const;** **int SlGrGridSelection::leftCol() const;** **int SlGrGridSelection::rightCol() const;**

表 4-145：***anchorRow / anchorCol***

anchorRow: 提供锚点单元格的行下标。 ***anchorCol:*** 提供锚点单元格的列下标。
int SlGrGridSelection::anchorRow() const; **int SlGrGridSelection::anchorCol() const;**

表 4-146：***numRows / numCols***

numRows: 提供选中区域中的行数。 ***numCols:*** 提供选中区域中的列数。
int SlGrGridSelection::numRows() const; **int SlGrGridSelection::numCols() const;**

表 4-147：***isActive / isEmpty***

isActive:告知，选中是否有效。 ***isEmpty:*** 告知，选中是否为空。
bool SlGrGridSelection::isActive() const; **bool SlGrGridSelection::isEmpty() const;**

4.8 菜单

术语解释

除了用于显示和输入数据的窗体外，屏幕还会在一条软键条(Softkey Bar)上显示多个软键这些软键会执行特定任务。软键条上软键的定义称为“菜单”。

屏幕上的每个软键条可以有多个菜单。在程序运行时可通过编程或定义在屏幕的这些菜单之间来回切换。

一个菜单可以包含多个菜单级，菜单级被称为 ETC 级。

菜单及其软键可以定义为“独有”。 在程序运行时不允许通过编程来创建新的菜单和软键，只允许修改现有软键。

XML 标签 MENU

在对话框配置文件中，菜单是用 XML 标签 MENU 来定义的。该标签通常是标签 SCREEN 的子标签，一些对话框通用的菜单除外，它们是标签 DIALOGUI 的子标签。

XML 标签 MENU 支持以下属性：

表 4-148：XML 标签 MENU

属性	描述
name	菜单的名称
softkeybar	需要显示菜单的软键条的名称。该名称和屏幕布局文件中为屏幕定义的软键条名称一致。
visible	指定在第一次显示屏幕时是否要显示菜单。 true（缺省）： 在显示屏幕时显示菜单。 false: 在显示屏幕时不显示菜单。 选择性指定
ref	对话框通用菜单的引用。此处要指定该菜单的名称（见“Attribut name”）。此时不允许设置所有其他属性。 选择性指定

示例：

```
<!DOCTYPE DIALOGUI>
<DIALOGUI defaultscreen="Screen1"
screenlayout="slstandardscreenlayout.SlStandardScreenLayout">
    <SCREEN name="Screen1">
        <FORM implementation="myform.MyForm" name="MyForm1"
                                             formpanel="FullForm"/>
        <MENU name="Horizontal1" softkeybar="hu">
            <!-- 软键的定义 -->
        </MENU>
        <MENU name="Horizontal2" softkeybar="hu" visible="false">
            <!-- 软键的定义 -->
        </MENU>
        <MENU name="Vertical" softkeybar="vr">
            <!-- 软键的定义 -->
        </MENU>
    </SCREEN>
</DIALOGUI>
```

本例中定义的“Screen1”包含三个菜单：“Horizontal1”、“Horizontal2”和“Vertical”。

菜单“Horizontal1”和“Horizontal2”显示在名为“hu”的软键条上， 而菜单“Vertical”显示在名为“vr”的软键条上。 这些软键条都在屏幕布局“SlStandardScreenLayout”中定义。 “hu”代表“Horizontal Upper”（水平居上），而“vr”代表“Vertical Right”（垂直居右）。

菜单“Horizontal1”和“Vertical”会在显示屏幕时一同显示。 菜单“Horizontal2”不一同显示，因 visible 设为“false”。

菜单级（ETC 级）

一个菜单可以包含多个菜单级，菜单级被称为 ETC 级。

ETC 级可以在定义菜单时加以定义。ETC 级的先后顺序和您定义时的顺序一致。

在对话框配置文件中，ETC 级是用 XML 标签 ETCLEVEL 来定义的。该标签是标签 MENU 的子标签。

XML 标签 ETCLEVEL 支持以下属性：

表 4-149：XML 标签 ETCLEVEL

属性	描述
id	ETC 级的名称。名称可以是文本也可以是数字。在菜单中该名称必须是唯一的。

示例：

```
<MENU name="hu" softkeybar="hu">
    <ETCLEVEL id="0">
        <!-- 软键的定义 -->
    </ETCLEVEL>
    <ETCLEVEL id="1">
        <!-- 软键的定义 -->
    </ETCLEVEL>
</MENU>
```

在本例中，菜单“hu”有两个 ETC 级，名为 ETC 级“0”和 ETC 级“1”。

可以在 HMI 对话框中使用的标准对话条有一个扩展按钮用于切换 ETC 级。每个操作面板上也有一个扩展键用于切换 ETC 级。 扩展按钮和扩展键的标记都为“>”。

在屏幕布局文件中定义了对话条中的扩展按钮和操作面板上的扩展键应作用于哪个软键条（见“屏幕布局”一章）。 在标准屏幕布局文件“SlStandardScreenLayout”中定义了菜单 ETC 级的扩展按钮显示在名为“hu”软键条中（hu 表示水平软键条），详见下文。

利用屏幕布局配置文件中的 XML 标签 ETCKEY 将软键条和扩展键关联起来。该标签是标签 SCREENLAYOUT 的子标签。

XML 标签 ETCKEY 支持以下属性：

表 4-150：XML 标签 ETCKEY

属性	描述
softkeybar	指定扩展键作用于哪个软键条。显示在该软键条上的 ETC 级可以利用该按键切换。

标准屏幕布局配置文件的选段：

```
<SCREENLAYOUT id="SlStandardScreenLayout" resolution="640x480">
    <SOFTKEYBAR id="hu" x="1" y="445" width="638" height="35"
orientation="Horizontal">
    ...
    </SOFTKEYBAR>
    ...
    <ETCKEY softkeybar="hu" />
    ...
</SCREENLAYOUT>
```

应用示例：
下面以定义一个有三个 ETC 级的菜单为例。 前提时相关屏幕使用了标准屏蔽布局，软键条“hu”中的菜单自动和扩展键关联。

对话条中的扩展按钮也指出了当前所处的 ETC 级。

```
<MENU name="hu" softkeybar="hu">
    <ETCLEVEL id="0">
        <!-- 软键的定义 -->
    </ETCLEVEL>
    <ETCLEVEL id="1">
        <!-- 软键的定义 -->
    </ETCLEVEL>
    <ETCLEVEL id="2">
        <!-- 软键的定义 -->
    </ETCLEVEL>
</MENU>
```

在本例中定义的水平软键条上的菜单有三个 ETC 级。按下操作面板上的扩展键或者按下对话条中的扩展按钮后，会依次切换 ETC 级。

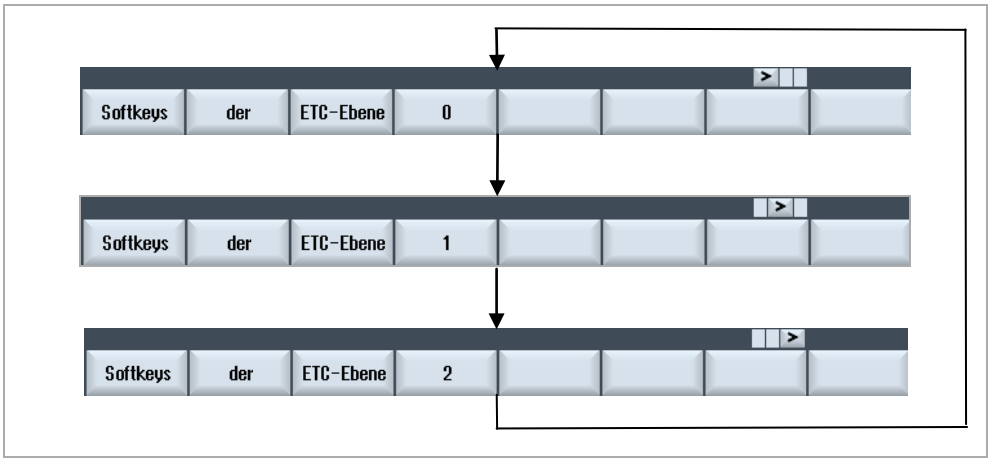


图 4-10:ETC 级的切换方式

ETC 级的切换有预定义的软键（见章节 4.9）和系统功能可用，您可以通过定义或编程来调用这些软键和功能（见章节 4.11）。

对话框通用的菜单

如果某个对话框的多个屏幕要使用相同的软键条菜单，您可以将该菜单定义为“对话框通用”，然后在各个屏幕中引用该菜单。
这种菜单的定义方式和普通菜单一样，只是普通菜单在标签 SCREEN 下定义，它在标签 DIALOG 下定义。 之后在需要使用该菜单的标签 SCREEN 下插入一个 <MENU>标签，该标签只具有属性：ref。 该属性的值为通用菜单的名称。

下面以定义一个名为“GlobalMenu”的通用菜单为例，该菜单同时用于“Screen1”和“Screen2”。

```
<DIALOGUI defaultscreen="Screen1"
screenlayout="slstandardscreenlayout.SlStandardScreenLayout">
    <MENU name="GlobalMenu" softkeybar="hu">
        <!-- ... 软键的定义 ... -->
    </MENU>
    <SCREEN name="Screen1">
        <!-- ... 窗体的定义 ... -->
        <MENU ref="GlobalMenu" />
    </SCREEN>
    <SCREEN name="Screen2">
        <!-- ... 窗体的定义 ... -->
        <MENU ref="GlobalMenu" />
    </SCREEN>
</DIALOGUI>
```

4.9 软键

术语解释

软键用于触发屏幕中的某项任务。一个软键相当于屏幕中的一个菜单项。

软键可以触发两种任务：

- 浏览
  切换屏幕、切换对话框或者切换操作区域。
- 功能
  执行窗体、屏幕或者对话框中实现的功能。

您需要定义软键显示的文本/图标以及软键需要执行的任务。软键的外观可以在程序运行时修改。

编程包为某些标准任务（确认、接收和取消等）提供了预定义的软键，可简化您的定义工作，使用户界面的设计统一。

切换软键（ToggleSoftkey）指在不同状态下有不同外观和执行不同任务的软键，该属性可以通过“profile”加以定义。

软键可以归为一组，以便设置单选按钮属性。

XML 标签 SOFTKEY

软键是通过 XML 标签 SOFTKEY 定义的。该标签是标签 MENU 或者 ETCLEVEL 的子标签。
XML 标签 SOFTKEY 支持以下属性：

表 4-151：XML 标签 SOFTKEY

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定可以查看和操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定

软键的所有其他属性，比如：软键上显示的文本通过属性指定，即使用 XML 标签 PROPERTY。

可用的属性是和使用的软键类型相关的。HMI 标准软键提供以下属性：

表 4-152：软键属性

属性	描述
id	可任意指定的软键文本，用于明确标识软键。只有需要在调试时设置软键的访问级时，才需要指定该属性。 （见调试手册章节 7.7“访问级”） 选择性指定 属性的数据类型：QString

属性	描述
textID	指定软键上显示的文本的 ID。 （没有指定 ID，软键上显示 textID） “%n”为换行符。 属性的数据类型：QString
translationContext	*textID* 的上下文。 没有指定该上下文时，访问文本时使用对话框的上下文。 选择性指定 属性的数据类型：QString
picture	需要在软键上显示的图标。此处应指定包含图标的文件的名称。 选择性指定 属性的数据类型：QPixmap
pictureAlignment	指定 *picture* 属性中指定的图标的对齐方式。允许的值有： AlignCenter 图标在软键上水平居中并垂直居中。 AlignLeft（缺省） 图标和软键的左边缘对齐，垂直居中。 AlignRight 图标和软键的右边缘对齐，垂直居中。 AlignTop 图标和软键的上边缘对齐，水平居中。 AlignBottom 图标和软键的下边缘对齐，水平居中。 选择性指定 属性的数据类型：Alignment
textAlignedToPicture	指定软键上显示的文本的输出位置。 true: 软键上有一个图标时，软键文本在剩余位置上居中输出。软键上没有图标时，文本居中输出。 false: 不管软键上是不是有图标，软键文本都居中输出。可能有的图标和文本有重叠时，软键文本叠在图标上显示。 属性的数据类型：bool
disabledFace	指定在“disabled”状态下如何显示软键。该属性的值可以为： Grayed: 在“disabled”状态下，软键文本灰显。 Empty: 在“disabled”状态下，软键文本以及可能有的图标被删除，不再显示。 用 HMI 的语言来说就是“该软键不再存在”。 该属性不能和属性“accesslevel”组合使用，换句话说，没有设置所需的访问级时都不显示软键，不管“disabledFace”设置如何。 属性的数据类型：SoftKeyDisabledFace

示例：

```
<SOFTKEY position="1" accesslevel="5">
    <PROPERTY name="textID" type="QString">MY_SOFTKEY_TEXT</PROPERTY>
    <PROPERTY name="picture" type="QPixmap">mypicture.png</PROPERTY>
    <PROPERTY name="pictureAlignment" type="Alignment">AlignTop</PROPERTY>
    <PROPERTY name="textAlignedToPicture" type="bool">true</PROPERTY>
    <!-- ... 软键任务 ... -->
</SOFTKEY>
```

本例中的软键显示在软键条的位置 1 上。 从访问级“钥匙开关 2”起显示该软键。 它显示 TextID 为 MY_SOFTKEY_TEXT 的文本以及文件 mypicture.png 中包含的图标。图标和软键的上边缘对齐显示，文本和图标对齐，显示在图标下方。

切换软键

切换软键（Toggle softkey）是可以有不同状态的软键。每种状态由一个“profile”加以说明，它定义了软键在该状态下的属性和任务。每次按下该软键都会切换状态。

状态的切换顺序是在配置文件中指定状态所属“profile”的顺序。

切换软键利用 XML 标签 TOGGLESOFTKEY 和 PROFILE 定义：

```
<TOGGLESOFTKEY 切换软键属性>
  <PROFILE profile 属性/>
    <PROPERTY name="propertyname" type="propertytype">value</PROPERTY>
      ... 更多软键属性的定义
      <FUNCTION 功能属性 />
        ... 功能标签
      </FUNCTION>
  </PROFILE>
  ... 更多 profile 的定义
</TOGGLESOFTKEY>
```

XML 标签 TOGGLESOFTKEY 支持以下属性：

表 4-153：XML 标签 TOGGLESOFTKEY

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定

切换软键的其他属性及其任务在其 profile 中确定。一个 profile 可以定义一种状态下软键的属性和任务。profile 是通过 XML 标签 PROFILE 定义的。

XML 标签 PROFILE 支持以下属性：

表 4-154：XML 标签 PROFILE

属性	描述
name	profile 的名称

示例：
下面以定义一个包含三个 profile 的切换软键为例。按下该软键后，会执行当前 profile 的任务，并切换到下一个 profile。

```
<TOGGLESOFTKEY position="1">
    <PROFILE name="Profile1">
        <PROPERTY name="textID" type="QString">Profile 1</PROPERTY>
        <!-- ... Profile1 的任务 ... -->
    </PROFILE>
    <PROFILE name="Profile2">
        <PROPERTY name="textID" type="QString">Profile 2</PROPERTY>
        <!-- ... Profile2 的任务 ... -->
    </PROFILE>
    <PROFILE name="Profile3">
        <PROPERTY name="textID" type="QString">Profile 3</PROPERTY>
        <!-- ... Profile3 的任务 ... -->
    </PROFILE>
</TOGGLESOFTKEY>
```

软键的切换顺序如下：

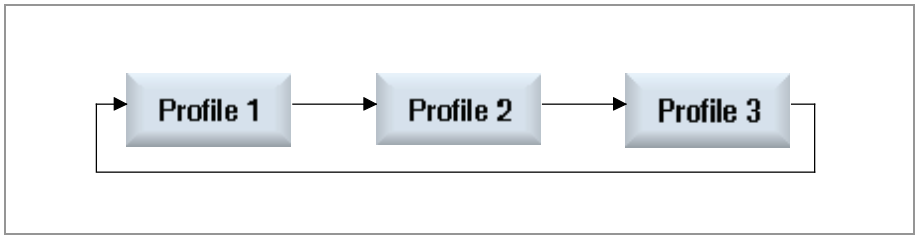


图 4-11:切换软键示例

- 在初始状态下，软键显示“Profile1”。
- 软键被按下。
- 执行“Profile1”的任务。
- 软键显示“Profile2”。
- 软键被按下。
- 执行“Profile2”的任务。
- 软键显示“Profile3”。
- 软键被按下。
- 执行“Profile3”的任务。
- 软键再次显示“Profile1”。

在程序运行时可以查询切换软键当前激活的的 profile，并激活特定 profile。

为此 SIGfwScreen 类提供两种方法：

```
QString currentToggleSoftkeyProfile(const QString& rszMenu,
                                    const QString& rszEtcLevel,
                                    unsigned int nPos) const
```

表 4-155：currentToggleSoftkeyProfile 的参数

属性	描述
rszMenu	切换软键所处的菜单的名称。
rszEtcLevel	切换软键所处的 ETC 级的名称。 没有定义 ETC 级时，指定 QString::null。
nPos	切换软键在菜单中的位置
返回值	当前激活的 profile 的名称

该方法返回切换软键当前显示的 profile 的名称。

```
int setCurrentToggleSoftkeyProfile(const QString& rszMenu,
                                   const QString& rszEtcLevel,
                                   unsigned int nPos,
                                   const QString& rszProfile);
```

表 4-156：setCurrentToggleSoftkeyProfile 的参数

属性	描述
rszMenu	切换软键所处的菜单的名称。
rszEtcLevel	切换软键所处的 ETC 级的名称。 没有定义 ETC 级时，指定 QString::null。
nPos	切换软键在菜单中的位置
rszProfile	要激活的 profile 的名称
返回值	故障代码

该方法选中切换软键的一个 profile。软键可由菜单、ETC 级和位置指定。如果 rszMenu 指定的菜单没有定义 ETC 级，将 rszEtcLevel 设为 QString::null。

Set 方法也作为系统功能提供。也就是说，该方法也可以定义为软键的功能。此时，参数在参数字符串中传递。

```
<FUNCTION name="setCurrentToggleSoftkeyProfile" args="-menu vr -etclevel 0
-position 1 -profile Profile2" />
```

一个展示切换软键设计的示例位于 GUIFrameWork\SIExGuiToggleSoftkey 的示例目录下。

软键“OK”

标准软键“OK”的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。

表 4-157： 标准软键“OK”

外观	标准任务
	该软键浏览到一个动态目标。其中传送的参数字符串包含参数**-sl_gfw_ok**。

软键“OK”的定义采用单独的 XML 标签 SOFTKEY_OK。该标签支持以下属性：

表 4-158：XML 标签 SOFTKEY_OK

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
args	指定在浏览时需要传送的附加参数。 选择性指定
defaultaction	指定软键是否执行标准任务。 true:浏览到一个动态目标（缺省）。 false:不浏览到一个动态目标。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_OK position="1" />
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。 比如：下面的软键显示文本“Yes”，但不执行浏览而是执行功能“SaidYes”。软键上有图标。在指定了 TextID 时还必须一同指定文本的上下文。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_OK position="1" defaultaction="false">
            <PROPERTY name="textID" type="QString">Yes</PROPERTY>
            <PROPERTY name="translationContext"
                      type="QString">MY_CONTEXT</PROPERTY>
            <FUNCTION name="SaidYes" />
        </SOFTKEY_OK>
    </MENU>
</SCREEN>
```

软键“Apply”

标准软键“Apply”的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。

表 4-159： 标准软键“Apply”

外观	标准任务
	该软键调用包含参数**-sl_gfw_apply** 的功能 onApply。

软键“Apply”的定义采用单独的 XML 标签 SOFTKEY_APPLY。该标签支持以下属性：

表 4-160：XML 标签 SOFTKEY_APPLY

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
args	在调用 onFunction()时传送的附加参数 选择性指定
defaultaction	指定软键是否执行标准任务。 true: 调用功能“onApply”（缺省）。 false: 不调用功能“onApply”。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。 屏幕中的窗体、屏幕本身或对话框必须在 onFunction 方法中对功能“onApply”作出响应。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_APPLY position="1" />
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。 比如：下面的软键显示文本“Yes”，但不执行浏览而是执行功能“SaidYes”。软键上有图标。在指定了 TextID 时还必须一同指定文本的上下文。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_APPLY position="1" defaultaction="false">
            <PROPERTY name="textID" type="QString">Yes</PROPERTY>
            <PROPERTY name="translationContext"
                      type="QString">MY_CONTEXT</PROPERTY>
            <FUNCTION name="SaidYes" />
        </SOFTKEY_APPLY>
    </MENU>
</SCREEN>
```

软键“Cancel”

标准软键“Cancel”的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。

表 4-161： 标准软键“Cancel”

外观	标准任务
	该软键浏览到一个动态目标。其中传送的参数字符串包含参数-**sl_gfw_cancel**。

软键“Cancel”的定义采用单独的 XML 标签 SOFTKEY_CANCEL。该标签支持以下属性：

表 4-162：XML 标签 SOFTKEY_CANCEL

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
args	指定在浏览时需要传送的附加参数。 选择性指定
defaultaction	指定软键是否执行标准任务。 true:浏览到一个动态目标（缺省）。 false:不浏览到一个动态目标。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_CANCEL position="1" />
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。 比如：下面的软键显示文本“No”，但不执行浏览而是执行功能“SaidNo”。软键上有图标。在指定了 TextID 时还必须一同指定文本的上下文。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_CANCEL position="1" defaultaction="false">
            <PROPERTY name="textID"
                      type="QString">No</PROPERTY>
            <PROPERTY name="translationContext"
                      type="QString">MY_CONTEXT</PROPERTY>
            <FUNCTION name="SaidNo" />
        </SOFTKEY_CANCEL>
    </MENU>
</SCREEN>
```

软键“Back”（浏览）

用于从子画面中返回的软键“Back”的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。该软键通常和软键“More”（带文本信息的菜单）一起使用。

表 4-163： 标准软键“Back”（浏览）

外观	标准任务
	该软键浏览到一个动态目标。

软键“Back”的定义采用单独的 XML 标签 SOFTKEY_NAV_BACK。该标签支持以下属性：

表 4-164：XML 标签 SOFTKEY_NAV_BACK

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
args	指定在浏览时需要传送的附加参数。 选择性指定
defaultaction	指定软键是否执行标准任务。 true:浏览到一个动态目标（缺省）。 false:不浏览到一个动态目标。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY_NAV_BACK position="1" />
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。 比如：下面的软键显示文本“Close”，但不执行浏览而是执行功能“onCloseMyScreen”。软键上有图标。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <SOFTKEY NAV BACK position="1" defaultaction="false">
            <PROPERTY name="textID" type="QString">Close</PROPERTY>
            <PROPERTY name="translationContext"
                      type="QString">MY CONTEXT</PROPERTY>
            <FUNCTION name="onCloseMyScreen" />
        </SOFTKEY NAV BACK>
    </MENU>
</SCREEN
```

软键“More”（含文本信息的菜单）

用于切换到子菜单或子画面中的软键“More”的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。该软键通常和软键“Back”（浏览）一起使用。

表 4-165： 标准软键“More”（含文本信息的菜单）

外观	标准任务
	切换到软键所处菜单的下一个 ETC 级。

软键“More”的定义采用单独的 XML 标签 SOFTKEY_MORE_TEXT。该标签支持以下属性：

表 4-166：XML 标签 SOFTKEY_MORE_TEXT

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
menu	指定切换到的目标菜单。当需要切换到另一个菜单而不是另一个 ETC 级时进行指定。 选择性指定
etclevel	指定切换到的目标 ETC 级。当需要切换到本菜单的下一个 ETC 级或者需要切换到另一个菜单的某个 ETC 级时进行指定。 选择性指定
defaultaction	指定软键是否执行标准任务。 true:切换到下一个 ETC 级（缺省）。 false:不切换到下一个 ETC 级。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。此时按下该软键后，应从 ETC 级 1 切换到 ETC 级 2。 另外，软键文本为“More”。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <ETCLEVEL id="1">
            <SOFTKEY_MORE_TEXT position="1">
                <PROPERTY name="textID" type="QString">More</PROPERTY>
            </SOFTKEY_MORE_TEXT>
        </ETCLEVEL>
        <ETCLEVEL id="2">
            ...
        </ETCLEVEL>
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。 比如，在下面的软键设计中，按下软键会切换到菜单 menu2 的 ETC 级 3，软键文本为“Details”。

```
<SCREEN 屏幕属性>
    <MENU name="menu1">
        <SOFTKEY_MORE_TEXT position="1" menu="menu2" etclevel="3">
            <PROPERTY name="textID" type="QString">Details</PROPERTY>
        </SOFTKEY_MORE_TEXT>
    </MENU>
    <MENU name="menu2">
        <ETCLEVEL id="1">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="2">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="3">
            ...
        </ETCLEVEL>
    </MENU>
</SCREEN>
```

软键“More”（不含文本的菜单）

用于切换到子菜单或子画面中的软键的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。该软键通常和软键“Back”（菜单）一起使用。

表 4-167： 标准软键“More”（不含文本的菜单）

外观	标准任务
	切换到软键所处菜单的下一个 ETC 级。

软键“More”的定义采用单独的 XML 标签 SOFTKEY_MORE。该标签支持以下属性：

表 4-168：XML 标签 SOFTKEY_MORE

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
menu	指定切换到的目标菜单。当需要切换到另一个菜单而不是另一个 ETC 级时进行指定。 选择性指定
etclevel	指定切换到的目标 ETC 级。当需要切换到本菜单的下一个 ETC 级或者需要切换到另一个菜单的某个 ETC 级时进行指定。 选择性指定
defaultaction	指定软键是否执行标准任务。 true:切换到下一个 ETC 级（缺省）。 false:不切换到下一个 ETC 级。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。此时按下该软键后，应从 ETC 级 1 切换到 ETC 级 2。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <ETCLEVEL id="1">
            <SOFTKEY_MORE position="1" />
        </ETCLEVEL>
        <ETCLEVEL id="2">
            ...
        </ETCLEVEL>
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。但对标准任务进行修改更加方便。 比如，在下面的软键设计中，按下软键会切换到菜单 menu2 的 ETC 级 3。

```
<SCREEN 屏幕属性>
    <MENU name="menu1">
        <SOFTKEY_MORE position="1" menu="menu2" etclevel="3" />
    </MENU>
    <MENU name="menu2">
        <ETCLEVEL id="1">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="2">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="3">
            ...
        </ETCLEVEL>
    </MENU>
</SCREEN>
```

软键“Back”（菜单）

用于返回上级 ETC 级的软键的定义采用一个单独的 XML 标签，以便简化和统一该软键的定义。该软键通常和软键“More”（不带文本的菜单）一起使用。

表 4-169： 标准软键“Back”（菜单）

属性	描述
	切换到软键所处菜单的前一个 ETC 级。

软键“Back”的定义采用单独的 XML 标签 SOFTKEY_BACK。该标签支持以下属性：

表 4-170：XML 标签 SOFTKEY_BACK

属性	描述
position	指定软键的位置。软键位置决定了软键在软键条中对应的按钮的位置。
accesslevel	指定操作软键所需的最低访问级。 此处没有指定访问级时，软键始终可用，不受当前设置的访问级的影响。 选择性指定
menu	指定切换到的目标菜单。当需要切换到另一个菜单而不是另一个 ETC 级时进行指定。 选择性指定
etclevel	指定切换到的目标 ETC 级。当需要切换到本菜单的前一个 ETC 级或者需

属性	描述
	要切换到另一个菜单的某个 ETC 级时进行指定。 选择性指定
defaultaction	指定软键是否执行标准任务。 true:切换到前一个 ETC 级（缺省）。 false:不切换到前一个 ETC 级。 选择性指定

在最简单的设计中，无需逐个指定所有属性。该软键的外观因此如上图所示，执行标准任务。此时按下该软键后，应从 ETC 级 2 切换到 ETC 级 1。

```
<SCREEN 屏幕属性>
    <MENU 菜单属性>
        <ETCLEVEL id="1">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="2">
            <SOFTKEY_BACK position="1" />
        </ETCLEVEL>
    </MENU>
</SCREEN>
```

当然您也可以重新设计该软键，取消标准任务。此时您可使用所有标准软键提供的属性。但对标准任务进行修改更加方便。比如，在下面的软键设计中，按下软键会切换到菜单 menu1 的 ETC 级 2。

```
<SCREEN 屏幕属性>
    <MENU name="menu1">
        <ETCLEVEL id="1">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="2">
            ...
        </ETCLEVEL>
        <ETCLEVEL id="3">
            ...
        </ETCLEVEL>
    </MENU>
    <MENU name="menu2">
        <SOFTKEY_BACK position="1" menu="menu1" etclevel="2" />
    </MENU>
</SCREEN>
```

软键组（单选按钮“Radio button”属性）

多个软键可以归为一组，以设置单选按钮属性。

设置了单选按钮属性后，在一个软键组内只能有一个软键或者没有任何软键可以处于“选定”状态。 按下该组的另一个软键后，新软键被选定，该组中的所有其他软键未被选定。

软键组的定义通过 XML 标签<SOFTKEYGROUP>定义：

```
<SCREEN 屏幕属性>
    <MENU 软键条属性>
        <SOFTKEY 软键属性>
            ...
        </SOFTKEY>
        <SOFTKEYGROUP 软键组属性>
            <SOFTKEY 软键属性>
            ...
            </SOFTKEY>
            <SOFTKEY 软键属性>
            ...
            </SOFTKEY>
        </SOFTKEYGROUP>
    </MENU>
</SCREEN>
```

XML 标签 SOFTKEYGROUP 支持以下属性：

表 4-171：XML 标签 SOFTKEYGROUP

属性	描述
name	软键组的名称。
selectedsoftkey	指定软键组中首个被选定的软键的位置。 选择性指定
toggleselectedsoftkey	指定在再次按下软键组中已选定的软键时是否会撤销定。 true:再次按下已选定的软键时，撤销选定。 false:再次按下已选定的软键时不会撤销选定。 选择性指定

软键组是菜单通用的，换句话说，软键组可以覆盖一个菜单内的多个 ETC 级。为此您要在各个 ETC 级中定义相同的软键组名称。

预定义功能设有附加的执行条件，以针对软键组内软键的选定和撤销选定作出不同响应。 该执行条件在 XML 标签 FUNCTION 的属性“execcondition”内指定。

- **softkeyselected**
  选定软键。
- **softkeydeselectedbytoggle**
  再次按下软键后撤销选定。
- **softkeydeselectedbyselchange**
  按下同一软键组内的其他软键后撤销选定。
- **softkeydeselected**
  撤销软键选定。

在按下并松开软键组内的某个软键时，执行条件按照以下的顺序处理：

表 4-172：XML 标签 SOFTKEYGROUP

执行条件	软键
softkeypressed	新的被按下的软键
softkeydeselectedbytoggle softkeydeselectedbyselchange softkeydeselected	迄今为止选中的软键（有的话）
softkeyselected	新的选中的软键（有的话）
softkeyreleased	新的被按下的软键

示例 1：

```
<SOFTKEYGROUP name="Group1" selectedsoftkey="1">
    <SOFTKEY position="1">
        <PROPERTY name="textID" type="QString">Group1::SK1</PROPERTY>
        <!-- 软键的任务 -->
    </SOFTKEY>
    <SOFTKEY position="2">
        <PROPERTY name="textID" type="QString">Group1::SK2</PROPERTY>
        <!-- 软键的任务 -->
    </SOFTKEY>
    <SOFTKEY position="3">
        <PROPERTY name="textID" type="QString">Group1::SK3</PROPERTY>
        <!-- 软键的任务 -->
    </SOFTKEY>
</SOFTKEYGROUP>
```

在本例中定义了一个软键组，软键位于位置 1、位置 2 和位置 3 上。位置 1 上的软键被预先选定。该
按下软键后软键组的响应方式如下：

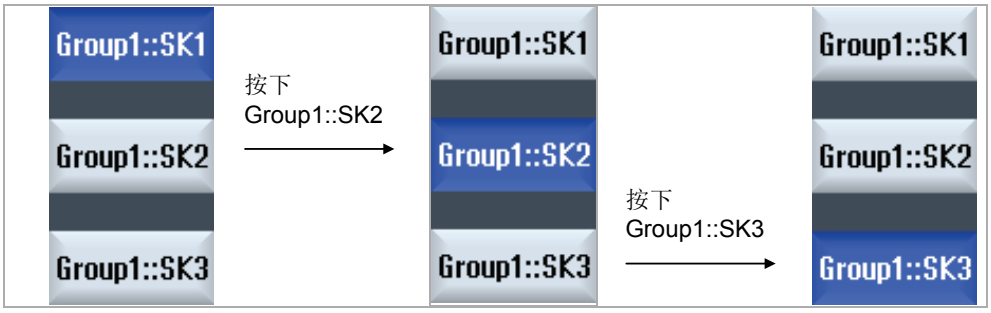


图 4-12:软键组

在程序运行时您可以查询并设置软键组中当前选定的的软键。为此 SIGfwScreen 类提供两种方法：

```
int selectedGroupSoftkey(const QString& rszMenu,
                         const QString& rszSoftkeyGroup,
                         QString& rszEtcLevel,
                         unsigned int& rnPos) const;
```

表 4-173：selectedGroupSoftkey 的参数

参数	描述
rszMenu	软键组所处菜单的名称。
rszSoftkeyGroup	被查询其状态的软键组的名称。
rszEtcLevel	该参数返回当前选定的软键所处的 ETC 级。
rnPos	该参数返回当前选定的软键在软键条中所处的位置。
返回值	故障代码。

该方法返回软键组中当前选定的软键所处的 ETC 级和位置。

```
int setSelectedGroupSoftkey(const QString& rszMenu,
                            const QString& rszSoftkeyGroup,
                            const QString& rszEtcLevel,
                            unsigned int nPos);
```

表 4-174：setSelectedGroupSoftkey 的参数

参数	描述
rszMenu	软键组所处菜单的名称。
rszSoftkeyGroup	软键组的名称。
rszEtcLevel	新选定的软键所处 ETC 级的 ID。
nPos	新选定软键所处的位置。
返回值	故障代码

该方法选定软键组中的一个软键。参数 rszMenu 和 rszSoftkeyGroup 指定软键组。ETC 级和位置指定软键。

如果 rszMenu 指定的菜单没有定义 ETC 级，将 rszEtcLevel 设为 QString::null。不需选定组中的任何软键时，将 rszEtcLevel 设为 QString::null，nPos 设为 0xFFFFFFFF。

通过编程来设置软键组中选定的软键时，也会执行在该软键上利用执行条件（softkeyselected、softkeydeselected、softkeydeselectedbytoggle 和 softkeydeselectedbyselchange）定义的所有功能。

Set 方法也作为系统功能提供。也就是说，该方法也可以定义为软键的功能。此时，参数在参数字符串中传递。

```
<FUNCTION name="setSelectedGroupSoftkey" args="-menu vr -softkeygroup
Group2 -etclevel 1 -position 7" />
```

本例选定的是软键组 Group2 中菜单 vr 内 ETC 级 1 位置 7 上的软键。
不需要选定组内任何一个软键时，删掉参数“-etclevel”和“-position”。

一个展示软键组设计的示例位于 GUIFrameWork\SIExGuiGroupSoftkey 的示例目录下。

软键的动态修改

有时您可能需要在程序运行时动态修改一个或者多个软键。该修改可以直接在软键的小部件上进行。

通过动态修改您可以重新定义软键的文本，激活或锁定软键，显示软键的“被按下”状态。但是这些修改不能持久，每次显示一个菜单时会丢失，必须重新执行。以这种方式激活了未定义的软键时，您必须在隐藏菜单后再次封锁该软键。

比较适合对软键进行动态修改的“地点”为虚拟方法 **onShowMenu()**和 **onHideMenu()**。这些方法位于类 IGfwHmiDialog 和 SIGfwHmiScreen 中。您可以从这些基本类导出自己的类，改写方法。

在 Framework 显示菜单**之后**，调用 onShowMenu()。
在 Framework 显示菜单**之前**，调 onHideMenu()。

这两种方法有以下参数：

表 4-175：onShowMenu()和 onHideMenu()的方法

参数	描述
rszMenu	菜单的名称。
rszEtcLevel	ETC 级的 ID。

屏幕类为访问软键小部件提供以下方法：

```
SlGfwSoftKey* softkey(const QString& rszSoftkeyBar,
                      unsigned int nPosition);
```

表 4-176：softkey()的参数

参数	描述
rszSoftkeyBar	屏幕布局文件中软键条的名称。
nPosition	软键在软键条上的位置。
返回值	软键小部件的指针。

该方法是访问软键的最快方法，但同时也是最不安全的方法。访问使用布局参数，即屏幕布局文件中软键条的名称和软键位置。您必须确认您需要修改的软键所处的菜单也在该软键条上显示。

```
SlGfwSoftKey* menuSoftkey(const QString& rszMenu,
                          unsigned int nPosition,
                          const QString& rszEtcLevel);
```

表 4-177：menuSoftkey()的参数

参数	描述
rszMenu	软键所处菜单的名称。
nPosition	软键在软键条上的位置。
rszEtcLevel	ETC 级的 ID（可选）
返回值	软键小部件的指针。

该方法使用定义的菜单的名称和软键位置来访问软键。菜单中当前不显示任何内容时，它返回 NULL 指针。可以额外指定 ETC 级用于附加检查。使用该方法时，您必须至少了解菜单中的哪个软键有特定功能。

```
SlGfwSoftKey* navigationSoftkey(const QString& rszScreen,
                                const QString& rszDialog,
                                const QString& rszArea,
                                const QString& rszMenu,
                                const QString& rszEtcLevel);
```

表 4-178：navigationSoftkey()的参数

参数	描述
rszScreen	屏幕的名称（屏幕浏览）
rszDialog	对话框的名称（对话框浏览）
rszArea	操作区域的名称（操作区域浏览）
rszMenu	菜单的名称
rszEtcLevel	ETC 级的 ID
返回值	软键小部件的指针。

利用该方法您可以请求一个指向软键小部件的指针，按下该软键后触发浏览。浏览目标通过前三个参数定义。

如果按下软键后应浏览到下一个屏幕，您需要指定参数 rszScreen。如果是浏览到下一个对话框，您要指定参数 rszDialog；浏览到下一个操作区域时指定 rszArea。没有指定参数时，指定 QString::null。rszDialog 和 rszArea 保持缺省值。

在调用方法时，GUI Framework 会检查菜单中是否有一个定义了相关浏览的软键，当前是否显示该菜单，检查完毕后返回小部件的指针。

您可以另外指定菜单和 ETC 级，使检查过程更加迅速。因此使用该方法时您无需了解哪个软键上定义了浏览。只要在另一个软键上定义了浏览，就无需修改源代码。

```
SlGfwSoftKey* functionSoftkey(const QString& rszFunction,
                              const QString& rszMenu,
                              const QString& rszEtcLevel);
```

表 4-179：functionSoftkey()的参数

参数	描述
rszFunction	函数名称
rszMenu	菜单的名称
rszEtcLevel	ETC 级的 ID
返回值	软键小部件的指针。

利用该方法您可以请求一个指向软键小部件的指针，按下该软键后触发特定功能。该功能在第一个参数中指定。

在调用方法时，GUI Framework 会检查菜单中是否有一个定义了相关功能的软键，当前是否显示该菜单，检查完毕后返回指向软键小部件的指针。

您可以另外指定菜单和 ETC 级，使检查过程更加迅速。因此使用该方法时您无需了解哪个软键上定义了功能。只要在另一个软键上定义了功能，就无需修改源代码。

```
QVariant softkeyProperty(const QString& rszMenu,
                         const QString& rszEtcLevel,
                         unsigned int nPos,
                         const QString& rszPropertyName);

bool setSoftkeyProperty(const QString& rszMenu,
                        const QString& rszEtcLevel,
                        unsigned int nPos,
                        const QString& rszPropertyName,
                        const QVariant& rvPropertyValue);
```

表 4-180：softkeyProperty()的参数

参数	描述
rszMenu	菜单的名称
rszEtcLevel	ETC 级的 ID
nPos	软键在软键条上的位置
rszPropertyName	属性的名称（对于软键文本而言是“textID”或“text”）
返回值	属性的值

表 4-181：setSoftkeyProperty()的参数

参数	描述
rszMenu	菜单的名称
rszEtcLevel	ETC 级的 ID
nPos	软键在软键条上的位置。
rszPropertyName	属性的名称（对于软键文本而言是“textID”或“text”）
rvPropertyValue	属性的值
返回值	当属性被成功修改时，返回“true”。

借助这两个方法您可以直接确定或修改软键的属性。如果软键的数据已知，就可以省略返回软键小部件指针这一中间步骤。

示例：修改软键文本

比如现在需要修改软键文本，便可以通过属性“textID”修改：

```
#include "slgfwhmidialog.h"
#include "slgfwsoftkey.h"

// new softkeytext or softkey-ID
QVariant newText("new SK-Text");

// get pointer to the softkey
SlGfwSoftKey* pSoftkey = dialog()->screenRef("screen")->menuSoftkey("vr", 1);

// set the new text
pSoftkey->setProperty("textID", newText);
```

也可以使用：

```
#include "slgfwhmidialog.h"
#include "slgfwscreen.h"

// get pointer to screen
SlGfwScreen* pScreen = dialog()->screenRef("myScreen");

// set the new text
pScreen->setSoftkeyProperty("vr", "etc_1", 4, "textID", "new SKText");
```

示例：将软键设为“disabled”

将软键设为“disabled”状态：

```
#include "slgfwhmidialog.h"
#include "slgfwsoftkey.h"

// get pointer to the softkey
SlGfwSoftKey* pSoftkey = dialog()->screenRef("screen")->menuSoftkey("vr", 1);

// set disabled
pSoftkey->setDisabled(true);
```

注

这些修改不能持久，每次在显示一个菜单时（比如：onShowMenu()）都要再次进行修改。

4.10 浏览

您可以为软键定义两种任务：

- 执行浏览
- 执行功能

本章向您介绍如何通过定义和编程来实现屏幕、对话框和操作区域的浏览。

XML 标签 NAVIGATION

软键的浏览可通过 XML 标签 NAVIGATION 定义，该标签是标签 SOFTKEY 的子标签。
XML 标签 NAVIGATION 支持以下属性：

表 4-182：XML 标签 NAVIGATION

属性	描述
target	指定浏览目标的类型。 *screen*: 浏览到一个屏幕。 *dialog*: 浏览到一个对话框。 *area*: 浏览到一个操作区域。 *dynamic*: 浏览到一个动态目标。
args	指定按下软键后向待选中的屏幕/对话框传递的执行参数。 选择性指定

目标屏幕、目标对话框或目标操作区域的名称在 NAVIGATION 的另一个子标签中指定。

浏览到一个屏幕

如需在按下软键后浏览到另一个屏幕，您必须在 XML 标签 NAVIGATION 中将参数 target 设为 screen。NAVIGATION 标签下随后有一个名为 SCREEN 的子标签，该标签可指定目标屏幕的名称。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="screen">
        <SCREEN name="TargetScreen" />
    </NAVIGATION>
</SOFTKEY>
```

在本例中，软键文本为“Target”。 按下该软键后，切换到屏幕“TargetScreen”。

XML 标签 SCREEN 支持以下属性：

表 4-183：XML 标签 SCREEN

属性	描述
name	目标屏幕的名称。

如需编程从一个对话框、屏幕或窗体到另一个屏幕的浏览，您可以调用对话框类中的方法 switchToScreen()。

```
int switchToScreen(const QString& rszScreen,
                   const QString& rszArgs);
```

表 4-184：switchToScreen()的参数

参数	描述
rszScreen	目标屏幕的名称。
rszArgs	指定向待选中的屏幕传递的执行参数。
返回值	故障代码。

示例：

```
// 调用一个对话框：
switchToScreen("TargetScreen", QString::null);

// 调用一个屏幕/窗体：
dialog()->switchToScreen("TargetScreen", QString::null);
```

区别是什么？

浏览到一个对话框

如需在按下软键后浏览到另一个对话框，您必须在 XML 标签 NAVIGATION 中将参数 target 设为 dialog。NAVIGATION 标签下随后有一个名为 DIALOG 的子标签，该标签可指定目标对话框的名称。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="dialog">
        <DIALOG name="TargetDialog" />
    </NAVIGATION>
</SOFTKEY>
```

希望直接跳转到目标对话框中的某个屏幕时，必须在 XML 标签 DIALOG 的参数 screen 中指定目标屏幕。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="dialog">
        <DIALOG name="TargetDialog" screen="TargetScreen" />
    </NAVIGATION>
</SOFTKEY>
```

XML 标签 DIALOG 支持以下属性：

表 4-185：XML 标签 DIALOG

属性	描述
name	目标对话框的名称。
screen	直接跳转到的目标对话框内的某个屏幕的名称。 选择性指定

如需编程从一个对话框、屏幕或窗体到另一个对话框的浏览，您可以调用对话框类中的方法 switchToDialog()。

```
int switchToDialog(const QString& rszDialog,
                   const QString& rszArgs,
                   bool bRemovePredecessor=false);
```

表 4-186：switchToDialog()的参数

参数	描述
rszDialog	目标对话框的名称。
rszArgs	指定向待选中的对话框传递的执行参数。
bRemovePredecessor	没有含义
返回值	故障代码。

示例：

```
// 调用一个对话框:
switchToDialog("ZielDialog", QString::null);

// 调用一个屏幕/窗体:
dialog()->switchToDialog("TargetDialog", QString::null);
```

希望直接跳转到目标对话框中的某个屏幕时，必须用第二个参数（执行参数）指定目标屏幕。 该参数名为“slGfwHmiScreen”。

```
// 调用一个对话框:
switchToDialog("TargetDialog", "-slGfwHmiScreen TargetScreen");

// 调用一个屏幕/窗体:
dialog()->switchToDialog("TargetDialog", "-slGfwHmiScreen TargetScreen");
```

浏览到一个操作区域

如需在按下软键后浏览到另一个操作区域，您可必须在 XML 标签 NAVIGATION 中将参数 target 设为 area。NAVIGATION 标签下随后有一个名为 AREA 的子标签，该标签可定义目标操作区域。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="area">
        <AREA name="TargetArea" />
    </NAVIGATION>
</SOFTKEY>
```

希望直接跳转到目标操作区域中的某个对话框时，必须在 XML 标签 AREA 的参数 dialog 中指定目标对话框。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="area">
        <AREA name="TargetArea" dialog="TargetDialog" />
    </NAVIGATION>
</SOFTKEY>
```

希望直接跳转到目标操作区域中的某个对话框和某个屏幕时，必须在 XML 标签 AREA 的参数 dialog 中指定目标对话框，在参数 screen 中指定目标屏幕。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="area">
        <AREA name="TargetArea" dialog="TargetDialog" screen="TargetScreen"
/>
    </NAVIGATION>
</SOFTKEY>
```

XML 标签 AREA 支持以下属性：

表 4-187：XML 标签 AREA

属性	描述
name	目标操作区域的名称
dialog	指定需要跳转到的目标操作区域中的对话框的名称。 选择性指定
screen	直接跳转到的目标对话框内的某个屏幕的名称。 选择性指定

如需编程从一个对话框、屏幕或窗体到另一个操作区域的浏览，您可以调用对话框类中的方法 switchToArea()。

```
int switchToArea(const QString& rszArea,
                 const QString& rszDialog,
                 const QString& rszArgs,
                 bool bRemovePredecessor=false);
```

表 4-188：switchToArea()的参数

参数	描述
rszArea	目标操作区域的名称
rszDialog	目标对话框的名称
rszArgs	指定向待选中的操作区域传递的执行参数。
bRemovePredecessor	没有含义
返回值	故障代码。

示例：

```
// 调用一个对话框:
switchToArea("TargetArea", QString::null, QString::null);

// 调用一个屏幕/窗体:
dialog()->switchToArea("TargetArea", QString::null, QString::null);
```

希望直接跳转到目标操作区域中的某个对话框时，必须用第二个参数指定目标对话框。

```
// 调用一个对话框:
switchToArea("TargetArea", "TargetDialog", QString::null);

// 调用一个屏幕/窗体:
dialog()->switchToArea("TargetArea", "TargetDialog", QString::null);
```

希望直接跳转到目标操作区域中的某个对话框和某个屏幕时，必须用第二个参数指定目标对话框，第三个参数指定目标屏幕。目标屏幕将传递到执行参数中。该参数名为“slGfwHmiScreen”。

```
// 调用一个对话框:
switchToArea("TargetArea", "TargetDialog", "-slGfwHmiScreen TargetScreen");

// 调用一个屏幕/窗体:
dialog()->switchToArea("TargetArea", "TargetDialog", "-slGfwHmiScreen
TargetScreen");
```

浏览到一个动态目标

每个软键都可以浏览到某个在系统运行时才确定的目标。这种浏览方式称为“浏览到一个动态目标”。

在调用一个屏幕时，动态目标可以作为执行参数传递给该屏幕。因此调用者本身也可以作为动态目标传递给被调用者，以便实现返回。
用于传递动态目标的执行参数为：

表 4-189：switchToArea()的参数

参数	描述
slGfwDynamicScreen	动态目标屏幕的名称
slGfwDynamicDialog	动态目标对话框的名称
slGfwDynamicArea	动态目标操作区域的名称

示例：

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">target</PROPERTY>
    <NAVIGATION target="screen" args="-slGfwDynamicScreen StartScreen">
        <SCREEN name="TargetScreen" />
    </NAVIGATION>
</SOFTKEY>
```

在本例中，按下软键后会浏览到屏幕“TargetScreen”。 “StartScreen”作为动态目标传递给“TargetScreen”。 如果“TargetScreen”包含了一个定义了动态浏览的软键，按下该软键后会返回到“StartScreen”。

您也可以通过编程来设置屏幕中的动态目标。为此可使用屏幕类中的方法 setDynamicTarget()。

```
void setDynamicTarget(const QString& rszScreen,
                      const QString& rszDialog,
                      const QString& rszArea);
```

表 4-190：setDynamicTarget()的参数

参数	描述
rszScreen	动态目标屏幕的名称
rszArea	动态目标操作区域的名称 选择性指定
rszDialog	动态目标对话框的名称 选择性指定

示例：

```
// 将屏幕设为返回目标
setDynamicTarget("Screen1");

// 将对话框设为返回目标
setDynamicTarget(QString::null, "Dialog1");

// 将操作区域设为返回目标
setDynamicTarget(QString::null, QString::null, "Area1");
```

如需在按下软键后浏览到一个动态目标，您可必须在 XML 标签 NAVIGATION 中将参数 target 设为 dynamic。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">DynamicTarget</PROPERTY>
    <NAVIGATION target="dynamic" />
</SOFTKEY>
```

如需编程从一个对话框、屏幕或窗体到一个动态目标的浏览，您可以调用对话框类中的方法 **switchToDynamicTarget()**。

```
int switchToDynamicTarget(const QString& rszArgs);
```

表 4-191：switchToDynamicTarget()的参数

参数	描述
rszArgs	指定向动态目标传递的执行参数。
返回值	故障代码。

示例：

```
// 调用一个对话框：
switchToDynamicTarget(QString::null);

// 调用一个屏幕/窗体：
dialog()->switchToDynamicTarget(QString::null);
```

返回键

每个操作面板都有一个返回键，而 HMI 对话框通常有的标准对话条中也有一个返回按钮。按键和按钮的标记都是“^”。

您同样可以任意定义返回键需要执行的浏览。为此可使用 XML 标签 RECALL。

XML 标签 RECALL 支持以下属性：

表 4-192：XML 标签 RECALL

属性	描述
accesslevel	指定操作软键所需的最低访问级。 选择性指定

返回键的标准任务是浏览到一个动态目标。只有在显示的屏幕中定义了返回键且为该屏幕设置了一个动态浏览目标时，返回键才可操作。

下面的示例定义了一个执行标准任务的返回键：

```
<SCREEN name="Screen1">
    <RECALL />
</SCREEN>
```

您也可以定义在按下返回键后跳转到某个目标，如同按下某个软键一样。为此您也可以使用标签 RECALL 下的子标签 NAVIGATION。

```
<SCREEN name="Screen1">
    <RECALL>
        <NAVIGATION target="screen">
            <SCREEN name="Screen2" />
        </NAVIGATION>
    </RECALL>
</SCREEN>
```

在本例中，按下返回键后浏览到屏幕“Screen2”。

对浏览的响应

在每次开始浏览前都会根据浏览方式调用以下对应的虚拟方法：

- onSwitchToScreen(const QString& rszScreen, QString& rszArgs, bool& rbDoIt)
- onSwitchToDialog(const QString& rszDialog, QString& rszArgs, bool& rbDoIt)
- onSwitchToArea(const QString& rszArea, const QString& rszDialog, QString& rszArgs, bool& rbDoIt)

这些方法既位于屏幕类中，也位于对话框类中。导出的类可以改写这些方法，以便对浏览作出响应。在开始浏览前，会首先调用当前显示的屏幕对象中的方法，然后调用对话框对象中的方法。

此时上述三个方法会传递一个字符串引用，该字符串包含了要传送给目标屏幕或目标对话框的执行参数(rszArgs)。因此无论是当前屏幕还是当前对话框都可以在开始浏览前修改或补充参数。

! **重要提示**

GUI Framework 会自行向参数字符串填入参数，因此字符串需要符合一定格式（见下文）。

另外，bool 型变量(rbDoIt)的引用会传递给这些方法。如果您将该布尔值设为 false，不会执行浏览。当前屏幕和对话框因此可以阻止浏览。

参数字符串的格式

GUI Framework 同样使用浏览方法的参数字符串，以便从一个屏幕向另一个屏幕传送参数。其中包含了在浏览期间使用必须符合一定格式的参数字符串。

该字符串的格式为 Name-Value-Pair。名称一般都以负号开头。名称后面是个空格，再跟一个数值。没有指定数值时，有参数名称这一状态会作为布尔值“true”传递。

```
-argumentname1 argumentwert1 -argumentname2
```

在本例中要传递一个名为“argumentname1”、值为“argumentwert1”的参数和一个名为“argumentname2”、值为“true”的参数。

需要传送一个空字符串时，可用一个占位符表示空字符串的参数值。占位符名为**SL_GFW_EMPTY_STRING**。

```
-argumentname1 SL_GFW_EMPTY_STRING
```

参数“argumentname1”现在包含一个空字符串。

GUI Framwork 提供所需方法用于正确编译或填入该格式的参数字符串。

用于参数字符串的辅助方法

SlGfwGuiComponent 类提供两个静态方法，利用这些方法您可以编译或定义参数字符串。在编译时字符串会转换为映射表。参数名称用作映射表的关键字以访问参数值。

方法 decodeArgString()对参数字符串进行解码：

```
static unsigned int decodeArgString(const QString& rszArgs,
                                    QMap<QString, QString>& rArgMap);
```

表 4-193：decodeArgString()的参数

参数	描述
rszArgs	参数字符串的引用。
rArgMap	包含了参数（“关键字-值”对）的映射表的引用
返回值	被解码的参数的数量。

示例：

```
#include "slgfwhmidialog.h"

QString szArgs = "-SearchString Axe -MatchCase -ReplaceString Achse";
QMap<QString, QString> argMap;
SlGfwGuiComponent::decodeArgString(szArgs, argMap);
```

在解码后映射表包含以下条目：

关键字	值
SearchString	Axe
MatchCase	true
ReplaceString	轴

通过映射表的运算符[]可以访问以下数值：

```
QString szSearchString = argMap["Searchstring"];
bool bMatchCase = argMap.contains("MatchCase");
QString szReplaceString = argMap["ReplaceString"];
```

方法 encodeArgString()对参数字符串进行编码：

```
static unsigned int encodeArgString(const QMap<QString, QString>& rArgMap,
                                    QString& rszArgs)
```

表 4-194：encodeArgString()的参数

参数	描述
rArgMap	包含了参数（“关键字-值”对）的映射表。
rszArgs	保存了参数的字符串的引用。
返回值	被编码的参数的数量。

示例：
假设以下值要作为参数传递：

关键字	值
MessageText	“An error occured!”
MessageIcon	“Error”

以下代码可用于填入映射表和对参数字符串进行编码：

```
QMap<QString, QString> argMap;
argMap["MessageText"] = “An error occured”;
argMap["MessageIcon"] = "Error";
QString szArgs;
SlGfwGuiComponent::encodeArgString(argMap, szArgs);
```

字符串 szArgs 接着包含以下内容：

```
“-MessageText An error occured!-MessageIcon Error”
```

encodeArgString()和 decodeArgString()是 SlGfwGuiComponent 类的两个静态方法，也就是说，在任何地方都可以调用这两个方法。
当然这两个方法也可以通过 SlGfwHmiDialog 类的静态方法调用或者直接通过对话框本身调用。

```
// 静态调用
SlGfwHmiDialog::encodeArgString(...);

// 直接调用
dialog()->encodeArgString(...);
```

参数的传送途径

在浏览多个方法时包含了执行参数的字符串会连续传递给方法。在每个此类方法中对象都可以修改参数字符串。

假设当前显示的是 ScreenA，现在需要切换到 ScreenB。此时参数字符串会传递给以下方法：

表 4-195：参数字符串的传送途径

方法	对象	描述
switchToScreen()	对话框	ScreenA 调用对话框方法 switchToScreen()，以切换屏幕。此时 ScreenA 可以传递一个参数字符串。
onSwitchToScreen()	ScreenA	接着 ScreenA 收到屏幕切换消息。ScreenA 此时可以再次修改参数字符串。
onSwitchToScreen()	对话框	对话框也收到屏幕切换消息。对话框此时也可以再次修改参数字符串。
close()	ScreenA	ScreenA 关闭。此时 ScreenA 有最后的机会可以再次修改参数字符串。
open()	ScreenB	在关闭 ScreenA 后 ScreenB 利用方法 open()打开。ScreenB 利用该方法获得参数字符串，现在可以评估该参数。
onScreenReached()	对话框	在显示 ScreenB 后，对话框得知该状态。此时它也会获得参数字符串。

通过参数字符串读出属性和传送属性

借助参数字符串可以在浏览时读出一个对象属性，并将它的值传送给浏览目标内某个对象的另一个属性。这种方式可以使两个对象之间的数据交换完全通过定义实现。对象可以是对话框、屏幕或窗体。因此在浏览时数据也可以直接传送给一个窗体，而不仅仅是传送给对话框或屏幕。

参数字符串中有一个命令可以触发两个对象通过属性交换数据。该命令的格式为“slgfwprop<No>”。<Nr>是一个流水号，因为在同一个参数字符串中您可能要指定多个属性进行数据传送。

在该命令后要指定目标对象内的目标属性，将源对象的源属性赋给目标属性。源属性必须是可以转换为字符串的数据类型，其中包括字符串属性、数字属性和枚举数属性。

```
-slgfwprop1 TargetObject.TargetProperty=SourceObject.SourceProperty
```

需要传送多个属性时，要调高命令末尾的编号。

```
-slgfwprop1
TargetObject1.TargetProperty1=SourceObjekt1.SourceProperty1
-slgfwprop2
TargetObject2.TargetProperty2=SourceObjekt2.SourceProperty2
```

此时，您可以为目标属性指定一个固定值。其中枚举数的值可以作为字符串指定。

```
-slgfwprop1 EditorForm.textAlignment=AlignCenter
```

示例：
在项目中有一个用于显示文件的 FileViewer 窗体和一个用于编辑文件的 Editor 窗体。FileViewer 窗体内有一个属性 currentItem，指出了当前选中的单元，而 Editor 窗体内有一个属性 fileName，指出了正在编辑器中编辑的文件的名称。
现在要在从包含了 FileViewer 窗体的屏幕浏览到包含了 Editor 窗体的屏幕时，currentItem 的值可以传送给 fileName，使 Editor 窗体打开 FileViewer 中选中的文件。
浏览的定义方式为：

```
<NAVIGATION target="screen" args="-slgfwprop1
EditorForm.fileName=FileViewerForm.currentItem">
    <SCREEN name="EditorScreen" />
</NAVIGATION>
```

GUI Framework 的参数

GUI Framework 有几个预定义的参数，可以在浏览时传送并用于修改需要显示的对话框或屏幕。

表 4-196：GUI Framework 的参数

参数	描述
slGfwHmiScreen	指定在显示对话框时需要显示的屏幕。 示例： "-slGfwHmiScreen MyScreen"
slGfwDynamicScreen	指定在下一屏幕中用作动态目标的屏幕。 示例： "-slGfwDynamicScreen MyScreen"
slGfwDynamicDialog	指定在下一屏幕中用作动态目标的对话框。 示例： "-slGfwDynamicDialog MyDialog"
slGfwDynamicArea	指定在下一屏幕中用作动态目标的操作区域。 示例： "-slGfwDynamicArea MyArea"
slGfwFocusForm	指定在显示屏幕时要获得操作焦点的窗体。 示例： "-slGfwFocusForm MyForm"
slGfwShowForm	指定在显示屏幕时需要显示的窗体。指定多个窗体时用逗号隔开。 示例： "-slGfwShowForm MyForm1,MyForm2"
slGfwShowMenu	指定在显示屏幕时要显示的菜单和 ETC 级。ETC 级的 ID 在菜单名称后的原括弧内指定。指定多个菜单时用逗号隔开。 示例： "-slGfwShowMenu FirstMenu,SecondMenu(2)" 在本例中显示菜单 FirstMenu 和 SecondMenu。在 SecondMenu 中显示 ETC 级 2。

4.11 功能

您可以为软键定义两种任务：

- 执行浏览
- 执行功能

本章向您介绍如何通过定义和编程在屏幕、对话框和窗体内执行功能。

功能的实现方式为：首先在 XML 标签中定义该功能，然后改写窗体、屏幕或对话框中的虚拟方法 onFunction()。

XML 标签 FUNCTION

您可以为每个软键定义一个或者多个需要执行的功能。功能可以任意多个，这些功能会按照您定义的先后顺序依次执行。为此您可以使用标签 SOFTKEY 下的子标签 FUNCTION。
XML 标签 FUNCTION 支持以下属性：

表 4-197：XML 标签 FUNCTION

属性	描述
name	需要执行的功能的名称。
args	功能参数。 选择性指定
execcondition	指定在何种条件下（即何时）需要执行功能。 *softkeypressed* 按下操作面板上的按键或点击鼠标左键按下软键后执行。 *softkeyreleased*（缺省） 松开操作面板上的按键或者松开鼠标左键，软键从被按下状态切换到松开状态后执行功能。 *softkeyselected* 选定了软键组中的某个软键时执行功能。 *softkeydeselectedbytoggle* 再次按下软键组中的某个已选定的软键来撤销选定时执行功能。 *softkeydeselectedbyselchange* 按下软键组中另一个软键来撤销之前选定的软键时执行功能。 *softkeydeselected* 撤销选定了软键组中的某个软键时执行功能。 选择性指定

在程序运行时按下软键后，经过定义的功能和参数会传递给虚拟方法 onFunction() 以进行编译。

借助执行条件可以在按下、松开、选定和撤销选定软键时使软键执行不同功能。

示例：
对话框实现了 setPLCBit 功能，即置位 PLC 中的一些位。现在在按下软键后，MB23.1 要置 1，松开软键后置 0。定义方式如下：

```
<SOFTKEY position="1">
    <PROPERTY name="text" type="QString">Do Sth.</PROPERTY>
    <FUNCTION name="setPLCBit" args="MB23.1, 1",
              execcondition="softkeypressed" />
    <FUNCTION name="setPLCBit" args="MB23.1, 0",
              execcondition="softkeyreleased" />
</SOFTKEY>
```

方法 onFunction()

对话框、屏幕和窗体基本类包含了一个虚拟方法，如需实现定义的功能，您必须改写该方法。该方法名为 onFunction()。

按下软键后会调用方法 onFunction()，以执行在 XML 标签 FUNCTION 下为该软键定义的功能。 该标签的属性（功能名称和参数字符串）会传送给该方法。根据这些数据可以在 onFunction()中接着调用特定功能或者直接在其中实现。

```
virtual void onFunction(const QString& rszFunction,
                        const QString& rszArgs,
                        bool& rbHandled);
```

表 4-198：onFunction()的参数

参数	说明
rszFunction	需要执行的功能的名称。 该名称是之前在 XML 标签 FUNCTION 中用属性“name”定义的名称。
rszArgs	功能参数。 该参数是之前在 XML 标签 FUNCTION 中用属性“args”定义的参数。
rbHandled	指出是否已经执行了功能（返回值）。

! **重要提示**

不管在哪个类中改写了 onFunction()，如果没有在 onFunction 实现中编辑经过定义的功能，您都要调用基本类的实现。

示例：
假设为软键定义了以下功能：

```
<FUNCTION name="setLabelText" args="Hello World!" />
```

另外，还有一个名为 MyForm 的窗体，它包含了一个名为 m_pMyLabel 的标签。在调用功能“setLabelText”时，作为参数传递的文本应写入该标签中。为此您要在窗体类中实现方法 onFunction()，该方法在评估参数后调用方法 setLabelText()：

```
void MyForm::onFunction(const QString & rszFunction,
                        const QString& rszArgs,
                        bool & rbHandled)
{
    if ("setLabelText" == rszFunction)
    {
        setLabelText(rszArgs);
        rbHandled = true;
    }
    else
    {
        SlGfwDialogForm::onFunction(rszFunction, rszArgs, rbHandled);
    }
}

void MyForm::setLabelText(const QString& rszText)
{
    m_pMyLabel->setText(szArgs);
}
```

首先在方法 onFunction()中检查它是否是已知方法，在本例中为“setLabelText”。如果是已知方法，则调用标签中包含了文本的功能。

需要注意的是，在成功执行功能后，参数 rbHandled 会置为“true”。用另一个功能名称调用 onFunction()时，该调用命令会传送给基本类，因为其中可能有实现了该功能的 GUI Framework 内置功能。

功能调用的过程

您可以在不同类中实现功能/方法 onFunction()：

- 对话框类
- 屏幕类
- 窗体类

按下一个定义了功能的软键后，该功能会依次提供该下述对象供执行（调用 onFunction()）：

- 当前显示的屏幕中具有操作焦点的窗体
- 当前显示的屏幕中其他可见窗体
- 当前显示的屏幕
- 对话框对象

在每次调用这些对象中的 onFunction()后，都会检查参数 rbHandled。 如果方法 onFunction()中的该参数置为“true”，即执行了功能，则终止将功能提供给其他对象。

但是您也可以通过指定所谓的 CommandHandler 只将功能提供给特定对象执行。

用于功能的 FunctionHandler

希望将定义好的功能只提供给某个预先确定的对象时，您可以在配置文件中通过另一个 XML 标签指定该对象。

功能可以提供给下述对象供执行：

- 定义了该功能的软键所在屏幕的所有窗体。
- 定义了该功能的软键所在的屏幕。
- 定义了该功能的软键所在屏幕的对话框。

您可以使用标签 FUNCTION 下的子标签 COMMANDHANDLER 指定对象。该标签支持以下属性：

表 4-199：XML 标签 COMMANDHANDLER

属性	描述
name	应执行功能的对象。
implementation	应执行功能的对象的实现。实现的说明格式为： library.classname

您不仅为 FunctionHandler 指定名称，也要指定实现（库和类名称）。

示例：

```
<SCREEN name="MyScreen"/>
    <FORM name="Form1" implementation="mylib.Form1" formpanel="LeftForm" />
    <FORM name="Form2" implementation="mylib.Form2" formpanel="RightForm"
/>
    <MENU name="MainMenu" softkeybar="hu">
        <SOFTKEY position="1">
            <FUNCTION name="myfunction">
                <COMMANDHANDLER name="Form1" implementation="mylib.Form1"
/>
            </FUNCTION>
        </SOFTKEY>
    </MENU>
</SCREEN>
```

在本例中，屏幕显示两个窗体：Form1 和 Form2。 水平软键条上有一个软键，按下该软键后执行功能“myfunction”。根据 CommandHandler 该功能只提供给 Form1 执行。

4.11.1 系统功能

基本类 SIGfwHmiDialog 和 SIGfwScreen 中已经实现了若干功能，您可以在定义时通过 XML 标签 FUNCTION 指定这些功能。此类功能称为“系统功能”。

您可以直接在基本类中调用系统功能或者通过 XML 标签 FUNCTION 在软键上定义。

表 4-200：系统功能

功能名称	类	描述
postMessage	SIGfwHmiDialog	向另一个 GUI 组件发送消息。
showForm	SIGfwScreen	显示指定的窗体。
hideForm	SIGfwScreen	隐藏指定的窗体。
showMenu	SIGfwScreen	显示指定 ETC 级和指定菜单。
showHelp	SIGfwScreen	显示在线帮助。
hideHelp	SIGfwScreen	隐藏在线帮助。
searchInHelp	SIGfwScreen	显示在线帮助查找。

系统功能 postMessage

系统功能 postMessage 可以向另一个 GUI 组件（比如：对话框）发送消息。此时该 GUI 组件也可以在另一个进程中运行。

该系统功能有三个参数。在定义时这三个参数字符串要用逗号隔开：

接收者
接收者的名称。此处 GUI 组件的名称必须和 systemconfiguration.ini 中用属性 name 确定的名称一致。
消息 ID
消息 ID 是 HMI 上唯一的编号，用于指出消息的类型。
消息数据
消息数据作为文本传送。

```
<FUNCTION name="postMessage" args="接收者,消息 ID,消息数据" />
```

在接收者对象中，一旦收到消息便调用虚拟方法 onMessage()。您可以改写该方法，以便定义收到特定消息时的响应。

示例：

```
<FUNCTION name="postMessage" args="MyDialog, 4711, these are data." />
```

系统功能 showForm

系统功能 showForm 可以显示一个屏幕中定义的所有窗体。

通常该功能用于那些属性 visible 设为 false 的窗体。 showForm 可以使这些原本“不可见”的窗体在按下软键后显示在界面上。这些窗体在参数字符串中指定，通过逗号隔开。

```
<FUNCTION name="showForm" args="Form1,Form2" />
```

如果某些窗体的属性 overlapping 没有设为 true，而新打开的窗体会部分或者完全覆盖它时，此类窗体自动隐藏。

示例：

```
<SCREEN name="MyScreen">
  <FORM implementation="mydialog.MyForm" name="MyFullForm"
formpanel="FullForm" />
  <FORM implementation="mydialog.MyForm" name="MyLeftForm"
formpanel="LeftForm" visible="false" />
  <FORM implementation="mydialog.MyForm" name="MyRightForm"
formpanel="RightForm" visible="false" />
  <MENU name="vr" softkeybar="vr">
    <SOFTKEY position="1">
      <PROPERTY name="textID" type="QString">Show Form 1</PROPERTY>
      <FUNCTION name="showForm" args="MyFullForm" />
    </SOFTKEY>
    <SOFTKEY position="2">
      <PROPERTY name="textID" type="QString">Show Form 2+3</PROPERTY>
      <FUNCTION name="showForm" args="MyLeftForm, MyRightForm" />
    </SOFTKEY>
  </MENU>
</SCREEN>
```

在本例中，屏幕中一共定义了三个窗体实例：MyFullForm、MyLeftForm 和 MyRightForm。 由于 MyLeftForm 和 MyRightForm 的属性 visible 设为 false，因此在显示屏幕时不显示这两个窗体。
另外，在该屏幕中还定义了两个软键。按下软键 1 后，显示窗体实例 MyFullForm。按下软键 2 后，显示窗体实例 MyLeftForm 和 MyRightForm。通过这种方式可以切换该屏幕的窗体，而无需切换屏幕。

系统功能 hideForm

方法 hideForm 可以在按下软键后隐藏当前显示的窗体。这些窗体在参数字符串中指定，通过逗号隔开。

```
<FUNCTION name="hideForm" args="Form1,Form2" />
```

示例：

```
<SCREEN name="MyScreen">
  <FORM implementation="mydialog.MyForm" name="MyFullForm"
formpanel="FullForm" />
  <FORM implementation="mydialog.MyForm" name="MyPopupForm"
formpanel="ModalForm" overlapping="true" visible="false" />
  <MENU name="vr" softkeybar="vr">
    <SOFTKEY position="1">
      <PROPERTY name="textID" type="QString">Show Popup</PROPERTY>
      <FUNCTION name="showForm" args="MyPopupForm" />
    </SOFTKEY>
    <SOFTKEY position="2">
      <PROPERTY name="textID" type="QString">Hide Popup</PROPERTY>
      <FUNCTION name="hideForm" args=" MyPopupForm" />
    </SOFTKEY>
  </MENU>
</SCREEN>
```

在本例的屏幕中，首先会显示窗体 MyFullForm。按下软键 1 后，显示窗体 MyPopupForm。由于该窗体的属性 overlapping 设为 true，因此它重叠在 MyFullForm 上显示。按下软键 2 后，再次隐藏窗体 MyPopupForm。

系统功能 showMenu

方法 showMenu 可以显示软键条上的某个菜单和某个 ETC 级。菜单和 ETC 级在参数字符串中指定，通过逗号隔开。

```
<FUNCTION name="showMenu" args="菜单名称,ETC 级" />
```

借助该方法您可以显示在 SCREEN 中属性 visible 设为 false 的菜单或者从当前菜单中切换到某个 ETC 级。

示例：

```
<SCREEN name="MyScreen">
  <MENU name="MyMenu" softkeybar="vr">
    <ETCLEVEL id="0">
      <SOFTKEY position="8">
        <PROPERTY name="textID" type="QString">More</PROPERTY>
        <FUNCTION name="showMenu" args="MyMenu,1" />
      </SOFTKEY>
    </ETCLEVEL>
    <ETCLEVEL id="1">
      <SOFTKEY position="8">
        <PROPERTY name="textID" type="QString">Back</PROPERTY>
        <FUNCTION name="showMenu" args="MyMenu,0" />
      </SOFTKEY>
    </ETCLEVEL>
  </MENU>
</SCREEN>
```

在本例的屏幕中，菜单“MyMenu”有两个 ETC 级。每个 ETC 级都有一个软键，按下该软键后切换到另一个 ETC 级。

系统功能 showHelp

系统功能 showHelp 可显示在线帮助。此时可以指定各种参数。

表 4-201：系统功能 showHelp 的参数

属性	描述
HelpDocument	需要显示在在线帮助中的帮助文件的名称。
HelpAnchor	帮助文件中需要跳转到的 HTML 锚点的名称。

参数要以相同的格式指定，如同浏览中的参数字符串。

没有指定参数时，该系统功能的作用如同操作面板上的信息键。帮助文件和锚点因此由虚拟方法 onHelp()从配置文件中确定，参见“slgfwdialogform.h”。

```
<FUNCTION name="showHelp" args="-HelpDocument hmi_myhelp/ chapter_1.html
                                -HelpAnchor />
```

系统功能 hideHelp

系统功能 hideHelp 可隐藏在线帮助。该方法没有参数。

```
<FUNCTION name="hideHelp" />
```

系统功能 searchInHelp

系统功能 searchInHelp 可以显示在线帮助并在其中开始查找。此时可以指定各种参数。

表 4-202：系统功能 searchInHelp 的参数

属性	描述
SearchText	查找关键字。
InSearchScreen	*true*: 显示查找结果屏幕。 *false*: 查找结果转化为一份 HTML 文档，在 HTML 浏览器中显示。(default) 选择性指定
FullTextSearch	*true*: 全字匹配查找。 *false*: 在目录/关键字目录中查找。(default) 选择性指定

参数要以相同的格式指定，如同浏览中的参数字符串。

```
<FUNCTION name="searchInHelp" args="-SearchText searchtext -InSearchScreen
false -FullTextSearch false" />
```

4.12 GUI 组件的属性

由于 GUI 组件（对话框、屏幕和窗体）的所有基本类都是从 Qt 类 QObject 中导出的，因此在这些类中您可以创建一些属性，这些属性可以在程序运行时通过定义对话框加动态设置。本章将向您介绍如何在一个类内创建属性，如何在配置文件中设置这些属性。

Qt 软件的属性

在从 QObject 导出的每个类中您都可以申明 Qt 属性。由于 GUI Framework 的所有基本类都是从 QObject 导出的，因此您可以在基本类中（比如窗体类）申明自定义的属性。

属性在类申明中申明。为此可使用宏命令 Q_PROPERTY。

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
            [RESET resetFunction] [DESIGNABLE bool]
            [SCRIPTABLE bool] [STORED bool] )
```

属性可以为所有数据类型，但要能保存到 QVariant 中。属性要使用类的 Qt-Meta-Object-System，因此类的头文件要由 MOC 编译器加以编译。

示例：
现在要使类获得名为 myBool 的一个布尔值属性。该属性应支持读/写。为此类申明应为：

```
class MyClass :public QObject
{
    Q_OBJECT
    Q_PROPERTY(bool myBool READ myBool WRITE setMyBool)
public:
    MyClass(QObject * pParent = 0, const char * szName = 0);
    ~MyClass(void);

    bool myBool(void) const;
    void setMyBool(bool bMyBool);
};
```

通过对话框配置文件来指定属性

在 GUI 类中完成了属性的申明和实现后，您也可以通过对话框的 XML 配置文件来指定该属性。此处以一个软键的定义为例。软键几乎所有的属性都可以定义，比如：软键上要显示的文本或图标。

需要向一个 GUI 对象（比如一个窗体或软键）指定属性时，可以使用标签 DIALOG、SCREEN、FORM 和 SOFTKEY 下的子标签 PROPERTY。

XML 标签 PROPERTY 支持以下属性：

表 4-203：XML 标签 PROPERTY

属性	描述
name	属性的名称。该名称和在宏命令 Q_PROPERTY 中作为第二个参数指定的名称一致。
type	属性的数据类型。目前您可以在配置文件中指定以下数据类型： *QString*: 单精度型字符串 *int*: 有符号整数 *uint*: 整整数 *double*: 浮点值 *bool*: 布尔值 *QRect*: 矩形（x，y，宽度，高度） *QSize*: 尺寸（宽度和高度） *QPoint*: 点(x, y) *QColor* : 颜色（RGB 或 HSV） *QDate*: 日期（年，月，日） *QTime*: 时间（小时，分钟，秒） *QDateTime*: 日期和时间。 *QPixmap*: 图片（图片的文件名） *QImage*: 图片（图片的文件名） *QBitmap*: 图片（图片的文件名） *Alignment*: Qt::Alignment Enum *SoftKeyDisabledFace*: SIGfwSoftKey::SoftKeyDisabledFace Enum

属性值一般在标签的文本区内指定，也就是开始标签和结束标签之间的区域。

有图片时一般指定图片的文件名称。在经过转换的配置文件中只能保存图片的文件名称。图片只有在程序运行时才载入。此时每个进程中载入一张图，以节省内存。

示例：
窗体 MyForm 包含了一个名为 listIndex 的属性，一个无符号整数值。在定义窗体时应该将该属性设为 7。定义方式应为：

```
<FORM implementation="mydialog.MyForm" name="MyForm" formpanel="FullForm">
      <PROPERTY name="listIndex" type="uint">7</PROPERTY>
  </FORM>
```

定义字符串型属性

```
<PROPERTY name="myProperty" type="QString">Hello!</PROPERTY>
```

定义一个 QString 型属性，值为“Hello!”。

定义整数型属性

```
<PROPERTY name="myProperty" type="int">-128</PROPERTY>
```

定义一个整数型属性，值为-128。

定义无符号整数型属性

```
<PROPERTY name="myProperty" type="uint">42</PROPERTY>
```

定义一个无符号整数型属性，值为 42。

定义浮点型属性

```
<PROPERTY name="myProperty" type="double">1.234567</PROPERTY>
```

定义一个浮点型属性，值为 1.234567。

定义布尔型属性

```
<PROPERTY name="myProperty" type="bool">true</PROPERTY>
```

定义一个布尔型属性，值为 true。

定义 QRect 型属性

```
<PROPERTY name="myProperty" type="QRect">
    <RECT x="2" y="4" width="63" height="169" />
</PROPERTY>
```

定义一个 QRect 型属性。矩形左上角的坐标为(2, 4)。矩形长 63 个像素，高 169 像素。

矩形通过 XML 标签 RECT 定义，该标签的属性有：

表 4-204：XML 标签 RECT

属性	描述
x	矩形左上角的 X 坐标。
y	矩形左上角的 Y 坐标。
width	矩形的宽度。
height	矩形的高度。

定义 QPoint 型属性

```
<PROPERTY name="myProperty" type="QPoint">
    <POINT x="47" y="11" />
</PROPERTY>
```

定义一个 QPoint 型属性。该点位于坐标(47, 11)上。
该点通过 XML 标签 POINT 定义，该标签的属性有：

表 4-205：XML 标签 POINT

属性	描述
x	点的 X 坐标。
y	点的 Y 坐标。

定义 QSize 型属性

```
<PROPERTY name="myProperty" type="QSize">
    <SIZE width="90" height="60" />
</PROPERTY>
```

定义一个 QSize 型属性。宽 90 个像素，高 60 个像素。

大小通过 XML 标签 SIZE 定义，该标签的属性有：

表 4-206：XML 标签 SIZE

属性	描述
width	宽度
height	高度

定义 QColor 型属性

```
<PROPERTY name="myProperty" type="QColor">
    <COLOR red="255" green="85" blue="6" />
</PROPERTY>

<PROPERTY name="myProperty" type="QColor">
    <COLOR hue="12" saturation="255" value="131" />
</PROPERTY>
```

定义一个 QColor 型属性。颜色可以指定为 RGB 值或者为 HSV 值。上述示例为相同颜色，即浅红色。

颜色通过 XML 标签 COLOR 定义，该标签的属性有：

表 4-207：XML 标签 COLOR

属性	描述
red	RGB 红色(0-255)
Green	RGB 绿色(0-255)
Blue	RGB 蓝色(0-255)
hue	HSV 颜色(0-255)
saturation	HSV 饱和度(0-255)
value	HSV 透明度(0-255)

定义 QDate 型属性

```
<PROPERTY name="myProperty" type="QDate">
    <DATE year="1977" month="7" day="21" />
</PROPERTY>
```

定义一个 QDate 型属性。其中包含的日期是 1977 年 7 月 21 日。

日期通过 XML 标签 DATE 定义，该标签的属性有：

表 4-208：XML 标签 DATE

属性	描述
year	四位数年份，如 2007。
month	月份，比如：11 代表十一月
day	日

定义 QTime 型属性

```
<PROPERTY name="myProperty" type="QTime">
    <TIME hour="20" minute="15" second="45" />
</PROPERTY>
```

定义一个 QTime 型属性。其中包含的时间为 20:15:45。

时间通过 XML 标签 TIME 定义，该标签的属性有：

表 4-209：XML 标签 TIME

属性	描述
hour	小时（0 到 23）。
minute	分钟（0 到 59）。
second	秒（0 到 59）。
millisecond	毫秒（0 到 99）。 选择性指定

定义 QDateTime 型属性

```
<PROPERTY name="myProperty" type="QDateTime">
    <DATETIME>
        <DATE year="1977" month="7" day="21" />
        <TIME hour="20" minute="15" second="45" />
    </DATETIME>
</PROPERTY>
```

定义一个 QDateTime 型属性。其中包含的日期是 1977 年 7 月 21 日，时间是 20:15:45。
日期时间的定义从 XML 标签 DATETIME 开始，该标签不包含任何属性。其中日期用子标签 DATE 定义，时间用字标签 TIME 定义。

定义 QPixmap 型属性

```
<PROPERTY name="myProperty"
type="QPixmap">mypicture.png</PROPERTY>
```

定义一个包含了图片“mypicture.png”的 QPixmap 型属性。在首次查询该属性的值时（QPixmap 对象），该图片只有在程序运行时才从文件中载入。

定义 QImage 型属性

```
<PROPERTY name="myProperty"
type="QImage">mypicture.png</PROPERTY>
```

定义一个包含了图片“mypicture.png”的 QImage 型属性。在首次查询该属性的值时（QImage 对象），该图片只有在程序运行时才从文件中载入。

定义 QBitmap 型属性

```
<PROPERTY name="myProperty"
type="QBitmap">mypicture.png</PROPERTY>
```

定义一个包含了图片“mypicture.png”的 QBitmap 型属性。在首次查询该属性的值时（QBitmap 对象），该图片只有在程序运行时才从文件中载入。

定义 Qt::Alignment 型属性

```
<PROPERTY name="myProperty"
type="Alignment">AlignCenter</PROPERTY>
```

定义一个 Qt::Alignment（枚举数）型属性，该属性值为 AlignCenter。

您可为该属性指定以下值：

- AlignLeft
- AlignRight
- AlignCenter
- AlignTop
- AlignBottom

定义 SlGfwSoftkey::SoftKeyDisabledFaceEnum 型属性

```
<PROPERTY name="disabledFace" type="SoftKeyDisabledFace">Grayed</PROPERTY>
```

定义一个 SlGfwSoftkey::SoftKeyDisabledFaceEnum（枚举数）型属性，该属性值为 Grayed。

您可为该属性指定以下值：

- Grayed
- Empty

4.13 与语言相关的文本

与语言相关的文本保存在语言文件中。一份语言文件只包含一种语言的文本。因此，您必须为每个 HMI 对话框每种语言创建一份语言文件。

语言文件中的单条文本组合在一起，构成“上下文”。一份语言文件可以包含多种上下文的文本。比如：您能可以将某个窗体的所有文本放置在一个上下文下。软键文本通常位于 HMI 对话框的上下文下。

TS 格式的语言文件

SINUMERIK Operate 使用和 Qt 类似的语言文件。只是它的 TS 格式增加了几个可选的 XML 标签。

TS 文件是一份 XML 文件，根是 XML 标签 TS。它的结构如下：

```
<!DOCTYPE TS>
<TS>
    <context>
        <name>context_name</name>
        <message>
            <source>text_id</source>
            <translation>待显示的文本</translation>
            <remark>Comment</remark>
            <chars>10</chars>
            <lines>2</lines>
        </message>
        ...
        更多消息
    </context>
    ...
    更多上下文
</TS>
```

该 XML 标签的含义如下：

表 4-210：TS 文件格式的 XML 标签

属性	描述
context	文件文件中的“语境”段。每个文件都必须至少有一个“语境”。
name	上下文的名称。
message	文本翻译。每个语境必须有一条消息。
source	文本 ID。
translation	翻译后的文本。
remark	文本的注释。 选择性指定
chars	指定允许的最大文本字符数量。如果没有指定，则文本可以为任意字符数。 选择性指定
lines	指定最多可用于显示的行数。如果没有指定，则为任意行数。 选择性指定

TS 文件是以 UTF-8 代码保存的。该代码只需在文本编辑器（比如：TextPad）的“另存为...”对话框中加以设置。UTF-8 代码可以保存一些特殊字符，比如：德语中的变音。

TS 文件的名称最好和所属的库名称一致。文件名由主名称、语言代码和扩展名“.ts”组成：主名称_语言代码.ts。比如：德语 TS 文件名为“mydialog_deu.ts”，英语 TS 文件名为“mydialog_eng.ts”。

特殊字符

TS文件中的特殊字符必须能够以UTF-8 代码保存。UTF-8 代码包含了大多数字母和字符，比如：德语中的变音、阿拉伯语、希腊语、俄语、韩语和泰语中的字符。您可以网上下载字符表http://www.utf8-zeichentabelle.de/。

在以 UTF-8 格式保存了 TS 文件后，您可以直接用德语键盘在该文件中写入德语变音。

但要输入一些属于 XML 命令的字符时（&、<、>和引号）要特别加以注意。在输入这些字符时要采用特殊格式，即以&开头，以;结尾。

表 4-211：XML 特殊字符

字符	TS 文件中的输入
&	&
<	<
>	>
“	"
'	'

当文本编辑器不支持一些字符时，采用这种格式您也可以输入这些无法通过键盘直接输入的字符。

比如：一个简单的文本编辑器是不支持希腊字符 Δ 的。查询 UTF-8 字符表后您可以发现，Δ 的字符代码是 0x0394。它的十进制值为 916，因此可以输入Δ来输入 Δ。

示例：
转换后的文本为：**ΔLänge X**
TS 文件中文本为：

```
<message>
  <source>DELTA LENGTH X</source>
  <translation>&#916;Länge X</translation>
</message>
```

TS 格式转换为 QM 格式

TS 格式的语言文件必须转换为二进制格式的 Qt 格式 QM，以适应 RUNTIME 环境。

为此您要将 TS 文件通过 WinSCP 传送到控制器。语言文件可以保存在以下目录中：
/oem/sinumerik/hmi/lng
/user/sinumerik/hmi/lng

在 HMI 重启时会自动将所有新的或更新过的 TS 文件转换为 QM 格式，然后提供给 RUNTIME 环境使用。

配置文件中的语言设置

HMI 对话框的语言文件在程序运行时动态装载和卸载。您可以在定义对话框时指定需要装载哪份 HMI 对话框语言文件。

XML 标签 DIALOGUI 为此提供两个属性：

- textfile
  指定语言文件，不含语言代码和扩展名。
- textcontext
  指定对话框的缺省上下文。所有没有明确指定上下文的文本都采用该上下文。

如果在该标签中只设置了 Textkontext，则该设置也同时适用于 HMI 对话框的所有屏幕和其中显示的软键。

```
<DIALOGUI defaultscreen="Screen1" textfile="mydialog"
textcontext="MyDialog">
   ...
</DIALOGUI>
```

您也可以为每个屏幕指定单独的上下文。XML 标签 SCREEN 为此也提供了 textcontext 属性。该屏幕的所有软键继承屏幕的上下文。

```
<SCREEN name="Screen1" textcontext="Screen1">
   ...
</SCREEN>
```

您也可以为每个软键指定单独的上下文。为此要设置软键属性 translationContext。同样也要设置属性 textID。

```
<SOFTKEY position="1">
    <PROPERTY name="textID" type="QString">MY_TEXT</PROPERTY>
    <PROPERTY name="translationContext"
type="QString">MY_CONTEXT</PROPERTY>
    <NAVIGATION target="screen">
        <SCREEN name="Screen1" />
    </NAVIGATION>
</SOFTKEY>
```

通过属性 textID 设置了软键文本后，切换语言时会自动读出该软键新语言的文本并将它显示在界面上。

窗体实现中的语言设置

窗体在国际化方面起特殊作用，这是因为窗体具有最大的重复使用潜力。窗体可以被另一个对话框重复使用，也可以作为小部件被另一个窗体使用。

出于上述原因，在实现窗体时就必须装载它的语言文件。为此可使用 SIGfwDialogForm 类的两个方法这两个方法在窗体的构造函数和析构函数中调用：

- setTextResource(const QString& rszTextFile, const QString rszTextContext)
  装载指定的语言文件（不含语言代码和扩展名），设置 readText()方法的缺省上下文。

- removeTextResource(const QString& rszTextFile)
  卸载指定的语言文件。

语言文件需要遵循“引用计数”规定，也就是说，如果一份语言文件被另一个对象（如窗体）使用，那么在调用 removeTextResource()时不会卸载该语言文件，只是引用计数器会减一。只有在计数器减为零后，才卸载语言文件。因此在构造函数 setTextResource()中调用窗体时也需要在析构函数 removeTextResource()中调用窗体，否则不会卸载语言文件。

读取语言文本

GUI 组件（对话框、屏幕和窗体）提供了方法 readText()用于读取语言文本。该方法收到一个 textID，有时也可能收到一个 textkontext，然后读取当前设置语言的文本，返回经过转换的文本。

readText()在读取文本时使用默认上下文。该上下文在定义对话框和屏幕时指定，在实现窗体时通过调用 setTextResource()指定。因此也可以在调用 readText()时将上下文作为第二个参数指定。

示例：
本示例将 ID 为 MY_LABEL_TEXT、转换后的文本设为“标签”。

```
m_pLabel->setText(readText("MY_LABEL_TEXT"));
```

本示例将 ID 为 MY_LABEL_TEXT、上下文为 MY_TEXT_KONTEXT、转换后的文本设为“标签”。

```
m_pLabel->setText(readText("MY_LABEL_TEXT", "MY_TEXT_KONTEXT"));
```

对语言切换的响应

在切换语言后，所有已装载的语言文件都会被新语言的文件替换。所有 GUI 组件随后收到通知“切换语言”。

GUI 组件提供一个槽，在切换语言后调用该槽，使各个对象可以更新自己的文本。槽的签名为：

```
virtual void onLanguageChanged(const QString& rszLanguage);
```

该槽在程序运行时会自动和一个信号关联起来，信号向所有可以对语言切换作出响应的对象通知这一事件。

在槽中，所有当前显示的文本必须再次借助 readText()重新读取和输出。该方法在生成 GUI 组件后也会被调用一次，以执行初始化。比如：对话框最好更新对话条上的输出。

通常只需要窗体对语言切换作出响应，更新文本。GUI Framework 会自动更新软键文本，前提是通过属性 textID 设置了该文本。

切换语言

对话框类提供两个方法，以进行语言切换：

```
long setLanguage(const QString& rszLanguage)
```

该方法用于确定当前语言。语言用语言代码指定，比如：deu, eng, fra。

```
long toggleLanguage(void)
```

您可以指定多种语言，在这些语言之间来回切换。该方法用于切换到下一个可切换语言，将该语言设为当前语言。

查询当前设置的语言

对话框类提供一个方法，用于查询当前设置的语言：

```
QString language(void) const
```

该方法返回当前设置的语言的代码。

其他语言设置

对话框类还提供了一些方法，用于查询或进行语言设置。

```
QStringList languages(void) const
```

返回一张指出当前所有安装语言的列表。列表中是语言的代码。

```
QStringList toggleLanguages(void) const
```

返回一张指出当前可切换语言的列表。列表中是语言的代码。这些语言是可以用 toggleLanguage()切换的语言。

```
long setToggleLanguages(const QStringList& rslLanguages)
```

设置一张指出可切换语言的列表。列表必须包含这些语言的代码。

示例

一个展示语言文本设计的示例位于 GUIFrameWork\SIExGuiLanguage 的示例目录下。

4.14 将在线帮助集成到系统中

一览

需要为 HMI 对话框设计在线帮助时，您可以采用 HelpService 的机制。设计在线帮助时，按下操作面板（TCU 或 OP）上的 INFO 键（帮助键）后，便可显示一份和语言相关的 HTML 格式的在线帮助文件。

设计示例位于 GUIFrameWork\SIExGuiOnlineHelp 的示例目录下。本例中的 HMI 对话框由两个 HMI 屏幕组成，通过软键可以来回切换屏幕。按下 INFO 键后在界面的下方会显示选定的屏幕对应的 HTML 帮助文件。再次按下 INFO 键后退出在线帮助。

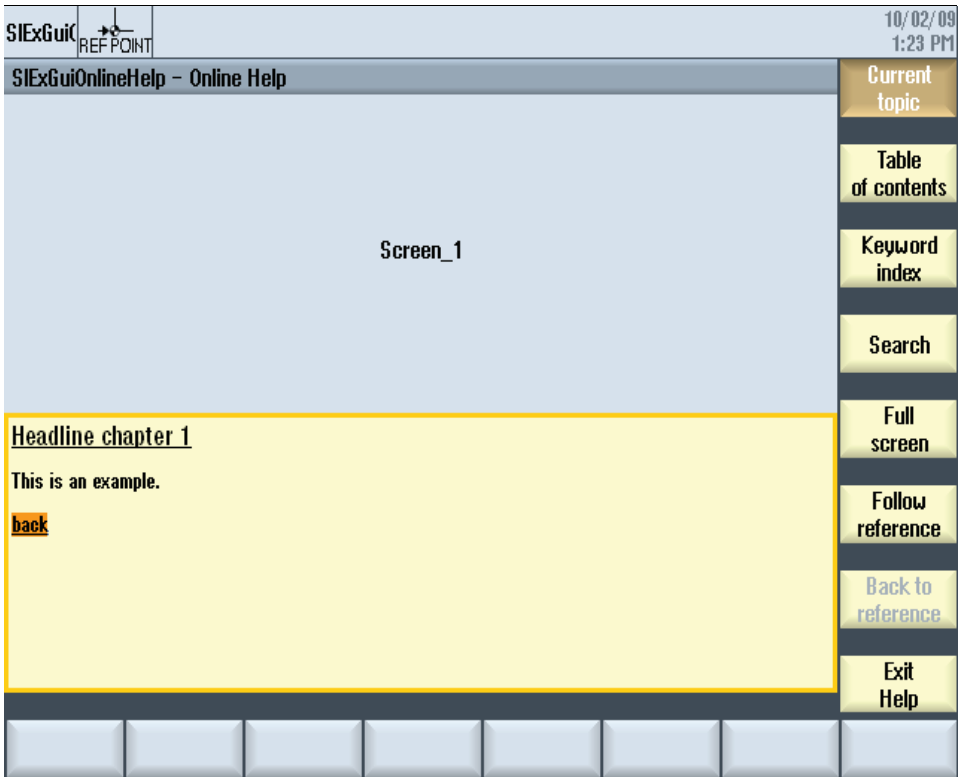


图 4-13:SIExGuiOnlineHelp

要实现这种在线帮助您需要完成以下步骤：

1. 创建帮助文件
2. 创建帮助手册（内容目录/关键词索引）
3. 将帮助手册集成到 SINUMERIK Operate 系统中
4. 复制创建的文件
5. 确定 HMI 对话框中的跳转点

下文将逐一介绍这些步骤。

创建帮助文件（HTML 文件）

帮助文件要以 HTML 格式创建。您可以将所有信息放在一份 HTML 文件中或者放在几份 HTML 文件中。文件名称您可以自由决定。这一步骤中要注意：

- HTML 文件内的引用要始终用绝对路径。只有这样才能保证引用不仅在开发计算机上有效，而且在载入系统后同样有效。
- 要稍后跳转到 HTML 文件中的特定点时，可以定义所谓的“锚点”。之后您就可以直接从 HMI 窗体跳到这个点。

  HTML 锚点示例：

```
<a name="myAnchor">This is an anchor</a>
```

- HTML文件的内容必须以UTF-8代码保存。只有这样才能确保HTML文件在所有SINUMERIK Operate支持的语言中都可正确显示。
- 如需了解支持的HTML标签，请查阅SINUMERIK Operate调试手册的章节17.2（HTML文件）。

创建帮助手册（目录/关键词索引）

帮助手册是一份描述了在线帮助结构的 XML 文件。它是目录和关键词索引的基础。

示例：一份含关键词的帮助手册(hmi_myhelp.xml)：

```
<?xml version="1.0" encoding="utf-8"?>
<HMI_SL_HELP language="en-US">
   <BOOK ref="index.html" title="My Help" helpdir="hmi_myhelp">
      <ENTRY ref="chapter 1.html" title="Chapter 1">
         <INDEX ENTRY ref="chapter 1.html#Keyword 1" title="Keyword 1"/>
         <INDEX_ENTRY ref="chapter_1.html#Keyword_2" title="Keyword_2"/>
      </ENTRY>
      <ENTRY ref="chapter_2.html" title="Chapter 2">
         <INDEX ENTRY ref="chapter 2.html#Keyword 3" title="Keyword 3"/>
      </ENTRY>
      <ENTRY ref="chapter 3.html" title="Chapter 3">
         <ENTRY ref="chapter_31.html" title="Chapter 31">
            <INDEX_ENTRY ref="chapter_31.html#test"
                                          title="test;chapter31"/>
         </ENTRY>
         <ENTRY ref="chapter 32.html" title="Chapter 32">
            <INDEX ENTRY ref="chapter 32.html#test"
                                          title="test;chapter32"/>
         </ENTRY>
      </ENTRY>
   </BOOK>
</HMI_SL_HELP>
```

此处显示的帮助手册描述了名为“My Help”的在线帮助。整个手册由 3 章组成，其中第三章包括两个子章节。在各章中均定义了不同的关键词。

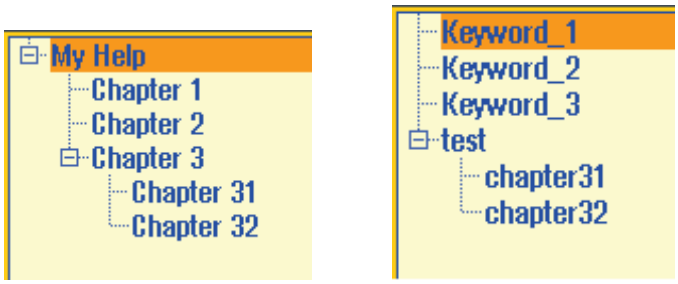


图 4-14:目录和关键词索引(hmi_myhelp.xml)

帮助手册的句法可以参见下表：

“数量”列：
*表示 0 或者更多
+表示 1 或者更多

表 4-212：帮助手册的格式说明

日	次数	含义	
HMI_SL_HELP	1	XML 文件的根单元	
I-BOOK I I I I .I I I I I	+	指定帮助手册	
		属性	
		ref	指定作为帮助手册的主页面显示的 HTML 文件。
		title	帮助手册的标题。它显示在目录中。
		helpdir	该帮助手册位于在线帮助中的目录。
I-ENTRY II II II II II II II	*	在线帮助的一个章节	
		属性	
		ref	指定作为本章的主页面显示的 HTML 文件。
		title	章节的标题。它显示在目录中。
II-INDEX_ENTRY II II II II II II II	*	待显示的关键词	
		属性	
		ref	指定该关键词跳转到的 HTML 文件。
		title	关键词条目的标题。它显示在关键词索引中。
II-ENTRY		见上文。	

您有三种方法可以设置关键词的格式。

格式只会影响某条关键词：

```
<INDEX_ENTRY ... title="index"/>
```

在该参数中加逗号可以创建两个两级索引，每个索引都有一条主条目和一条子条目。

```
<INDEX_ENTRY ... title="mainIndex_1,subIndex_1 with mainIndex_1"/>
```

在该参数中加分号可以创建一个两级索引，第一个是主条目的标题，第二个是子条目的标题。

```
<INDEX_ENTRY ... title="mainIndex_2;subIndex_2 without mainIndex_1"/>
```

图 4-15:不同的索引结构

将帮助手册集成到 SINUMERIK Operate 系统中

您需要使用文件“slhlp.xml”，将创建的帮助手册通知给中央 HelpService。

文件“slhlp.xml”示例

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE CONFIGURATION>
<CONFIGURATION>
   <OnlineHelpFiles>
      <hmi_myHelp>
         <EntriesFile value="hmi_myhelp.xml" type="QString"/>
         <DisableIndex value="true" type="bool"/>
      </hmi_myHelp>
   </OnlineHelpFiles>
</CONFIGURATION>
```

本例公布了帮助手册“hmi_myhelp.xml”，其中的关键词索引未激活。

文件“slhlp.xml”的句法可以参见下表。

“数量”列：
*表示 0 或者更多
+表示 1 或者更多

表 4-213：“slhlp.xml”文件的格式描述

日	次数	含义	
CONFIGURATION	1	XML 文件的根单元。用于标识配置文件。	
I-OnlineHelpFiles	1	引入在线帮助手册的段落。	
II-<help_book>	*	引入帮助手册的段落。	
III-EntriesFile III III III III III III	1	含目录条目和关键词条目的帮助手册的名称。	
		属性：	
		value	含目录条目和关键词条目的帮助手册的名称。
		type	数值的数据类型(QString)
III-Technology III III III III III III III III III III III	0,1	指定该帮助手册适用的加工工艺。“All”表示适用于所有工艺。如需手册适用于多个工艺，工艺之间要用逗号隔开。 允许的值： All, Universal, Milling, Turning, Grinding, Stroking, Punching	
		属性：	
		value	帮助手册适用的工艺。
		type	数值的数据类型(QString)
III-DisableSearch III III III III III	0,1	关闭帮助手册的关键词搜索	
		属性：	
		value	true, false
		type	数值的数据类型(bool)
III-DisableFullTextSearch III III III III	0,1	关闭帮助手册的全字匹配搜索	
		属性：	
		value	true, false
		type	数值的数据类型(bool)
III-DisableIndex III III III III III	0,1	关闭帮助手册的关键词索引	
		属性：	
		value	true, false
		type	数值的数据类型(bool)
III-DisableContent III III III III III	0,1		关闭帮助手册的目录
		属性：	
		value	true, false
		type	数值的数据类型(bool)
III III III III III III III	0,1	指定帮助手册显示语言的代码（如果有语言文件）。	
		属性：	
		value	Deu, eng, esp, fra, ita. ....
		type	数值的数据类型(QString)

复制创建的文件

现在您可以将创建好的文件复制到系统中：

1. 在路径“<安装路径>/oem/sinumerik/hmi/hlp”下创建一个目录。目录的名称要和语言代码一致。

   比如说，要集成德语版和英语版的帮助文件时，应创建以下目录：
   “<安装路径>/oem/sinumerik/hmi/hlp/deu”
   “<安装路径>/oem/sinumerik/hmi/hlp/eng”。

2. 将帮助手册保存到该目录下。

   在本例中保存的是德语版和英语版的帮助手册，手册的路径为：
   “<安装路径>/oem/sinumerik/hmi/hlp/deu/hmi_myhelp.xml”
   “<安装路径>/oem/sinumerik/hmi/hlp/eng/hmi_myhelp.xml”。

3. 将配置文件“slhlp.xml”复制到目录“<安装路径>/oem/sinumerik/hmi/cfg”中。为节省系统的存储空间，您可以利用工具“slHmiConverterGui”将该文件转换为“slhlp.cfg”，工具位于 SINUMERIK Operate 编程包的“Tools”中。

   在本例中这两种格式都可以：
   “<安装路径>/oem/sinumerik/hmi/cfg/slhlp.xml”
   或
   “<安装路径>/oem/sinumerik/hmi/cfg/slhlp.cfg”。

重启 HMI 后，您便可以从 HMI 对话框调用在线帮助。

注

在显示帮助手册的目录和关键词索引时，为了进行快速处理，目录**“<安装路径>/siemens/sinumerik/sys_cache/hmi/hlp”**下会生成一份二进制格式的帮助文件(slhlp_<Helpbook_*.hmi)。即使稍稍修改了帮助手册，也要删除这些文件。

确定 HMI 对话框中的跳转点

您还需要定义跳转点，才可以使用在线帮助。 跳转点可以在对话框配置文件的对话框段落、屏幕段落和/或窗体段落的属性 helpdocument 和 helpanchor 中定义。

另外，您还要将帮助的屏幕导入到对话框配置文件中。

含在线帮助的对话框配置文件的选段：

```
<!DOCTYPE HMI DIALOG CONFIGURATION>
<DIALOGUI ...
   helpdocument="hmi myhelp/chapter 1.html"
   helpanchor="myAnchor_1">

      <IMPORT file="Y:/gui/dialogs/common/slgfwcommon_incl.xml"
              screen="SlHlpBrowserScreen" />
      <IMPORT file="Y:/gui/dialogs/common/slgfwcommon incl.xml"
              screen="SlHlpFullScreen" />
      <IMPORT file="Y:/gui/dialogs/common/slgfwcommon_incl.xml"
              screen="SlHlpSearchScreen" />

      <SCREEN ...
             helpdocument="hmi myhelp/chapter 2.html"
             helpanchor="myAnchor_2">

        <FORM ...
              helpdocument="hmi myhelp/chapter 3.html"
              helpanchor="myAnchor 3"/>
   </SCREEN>
</DIALOGUI>
```

如果在安装时没有选择驱动盘“Y：”作为替代驱动盘，请做出相应调整。

注

跳转点的优先级从高到低为：窗体、屏幕和对话框。比如：同时定义了窗体中的跳转点和对话框中的跳转点时，窗体中的跳转点生效。

其他调用方式

除了通过 INFO 键激活在线帮助外，还可以通过另外两种机制激活在线帮助：

1. 通过系统功能“showHelp”（定义方式）：

```
<SOFTKEY ...>
<PROPERTY name="textID" type="QString">showHelp</PROPERTY>
<FUNCTION name="showHelp" args="-HelpDocument hmi_myhelp/chapter_1.html
                                -HelpAnchor myAnchor" />
</SOFTKEY>
```

→通过该方法您可以比如在按下一个软键后显示在线帮助。

2. 通过系统功能“showHelp”（编程方式）：

```
dialog()->showHelp("hmi_myhelp/chapter_1.html", "myAnchor");
```

→通过该方法您可以比如在输入一个无效值后显示在线帮助。

更多关于这两种方法的信息在章节 4.11.1“系统功能”中说明。

功能 onHelp()

对话框、屏幕和窗体的基本类都包含了一个虚拟功能，您希望动态修改属性 helpdocument 和 helpanchor 时必须改写该功能。

该功能的调用方式有：
- 按下 INFO 键
- 调用了没有参数的 showHelp

此时该功能会以此提供给下列对象供执行：
- 当前显示的屏幕中具有操作焦点的窗体
- 当前显示的屏幕中其他可见窗体
- 当前显示的屏幕
- 对话框对象

在每次调用这些对象中的 onHelp 后，都会检查参数 rbHandled。 如果该参数置为 “true”，即执行了功能，则终止将功能提供给其他对象。

onHelp()在显示帮助前调用。 通过改写参数 rszDocument 和 rszAnchor 您可以修改需要显示的帮助。

```
virtual void onHelp(QString& rszDocument,
                    QString& rszAnchor,
                    bool& rbHandled);
```

表 4-214：onHelp()的参数

参数	描述
rszDocument	待显示的在线帮助的名称
rszAnchor	指定从哪个 HTML 锚点开始显示 rszDocument 属性指出的在线帮助文件。
rbHandled	指出是否已经执行了功能（返回值）

! **重要提示**

不管在哪个类中改写了 onHelp()，如果没有在 onHelp 实现中编辑经过定义的功能，您都要调用基本类的实现。

机床数据的在线帮助

希望为特定机床数据显示自定义的在线帮助时，同样可以在一份 HTML 文件中写入帮助的内容，将该文件按照机床数据号命名，比如，14510.html 代表机床数据 14510 $MN_USER_DATA_INT 的帮助文件。

将该 HTML 文件保存到以下目录下：

NC 机床数据（比如 14510 $MN_USER_DATA_INT）：
“<安装路径>/oem/sinumerik/hmi/hlp/<语言>/sinumerik_md_nck/14510.html”

通道专用的机床数据（比如 27400 $MC_OEM_CHAN_INFO）：
“<安装路径>/oem/sinumerik/hmi/hlp/<语言>/sinumerik_md_chan/27400.html”

轴专用的机床数据（比如 37800 $MA_OEM_AXIS_INFO）：
“<安装路径>/oem/sinumerik/hmi/hlp/<语言>/sinumerik_md_axis/37800.html”

4.15 文件访问

SINUMERIK Operate 的文件有固定的目录结构。一级目录有规定的优先级，优先级确定了当一份文件位于多个目录中时装载哪个目录中的文件。

文件的查找范围覆盖了以下目录：

1. <安装路径>/**user**/sinumerik/hmi
2. <安装路径>/**oem**/sinumerik/hmi
3. <安装路径>/**addon**/sinumerik/hmi
4. <安装路径>/**siemens**/sinumerik/hmi

目录 user 的优先级最高，然后依次是目录 oem、addon 和 siemens。 首先在目录 user 中查找文件。 如果在其中没有找到文件，则接着在目录 oem 中查找，依此类推。

每个上述目录都包含了子目录，子目录中保存的特定数据：

- **appl**
  应用程序和库(exe, dll, so)
- **ico/ico<分辨率>**
  图标和图片(png, bmp)
- **lng**
  语言文件(qm, ts)
- **cfg**
  配置文件(ini, xml, cfg)

为按照上述目录优先级查找文件，程序提供有 SlHmiFileAccessQt 类。

SlHmiFileAccessQt

借助 SlHmiFileAcessQt 类的方法可以按照 SINUMERIK Operate 的指定目录优先级查找文件。

在以下功能中，子目录（如 appl, ico, lng, …）始终要在参数 rszSubPath 中传递，不允许一同写入文件名称中。

以下方法用于查找文件：

- searchFileLocations
- searchFileLocationsWithWildcards
- searchFile

以下方法用于查找子目录中的图标和图片：

- searchImageFileLocations
- searchImageFile

查找文件

查找文件需要生成 SlHmiFileAccessQt 类的一个实例。方法 **searchFile**、**searchFileLocations** 或 **searchFileLocationsWithWildcards** 用于在 HMI 目标路径中查找文件。

示例：

```
#include "slhmisettingsqt.h"

void SlBspClass::bspFunction(...)
{
    SlHmiFileAccessQt fileAccess;
    QString szPath;
    QString szLibName;
    long nError = 0;

#if defined(Q_WS_WIN)
    szLibName = "bsp.dll";
#else
    szLibName = "libbsp.so";
#endif

    nError = fileAccess.searchFile (szLibName, "appl", szPath);

    if(SL SUCCEEDED(nError))
    {
        QLibrary lib(szPath);
        ...
    }
}
```

在本例中要查找库 bsp.dll（也就是 Linux 系统下的 libbsp.so）。此类文件位于子目录 appl 中。

查找图片文件

查找特定分辨率的图标或图片需要生成 SlHmiFileAccessQt 类的一个实例。方法 **searchImageFile** 或 **searchImageFileLocations** 可用于在 HMI 目标路径中查找图片文件。

示例：

```
#include "slhmisettingsqt.h"

void SlBspClass::bspFunction(...)
{
    SlHmiFileAccessQt fileAccess;
    QString szPath;
    long nError = 0;
    QPixmap pixmap

    nError = fileAccess.searchImageFile("test.png",
                                        QString::null,
                                        640,
                                        szPath);

    if(SL SUCCEEDED(nError))
    {
        pixmap.load(szPath);
        ...
    }
}
```

在本例中要查找分辨率为 640x480 像素的图片 test.png。

4.16 配置数据（INI 文件）

HMI 应用程序可以将配置数据保存为文件格式，随后在程序运行时读取这些文件并作出相应的响应。配置文件中也可以保存一些需要断电保存的状态数据。

配置文件有三种格式：

- INI 文件
  Windows 系统的典型配置文件。
- XML 文件
  配置文件的 XML 格式，数据等级可以更深。
- CFG 文件
  二进制格式，其中保存了 INI 文件或 XML 文件的数据。

配置文件可以借助转换程序转换为其他格式。

程序运行时可以读取任何一种格式的配置文件。一个访问类实现了格式的转换。

配置文件始终保存在子目录 cfg 中，详细的路径为：

- **<安装路径>/user/sinumerik/hmi/cfg** 或
- **<安装路径>/oem/sinumerik/hmi/cfg** 或
- **<安装路径>/addon/sinumerik/hmi/cfg** 或
- **<安装路径>/siemens/sinumerik/hmi/cfg**

原则上在 siemens 目录中不允许保存任何配置文件，因为其中保存的是基系统的配置文件。

如果一份配置文件同时位于多个子目录中，在读取时会把这些文件整合在一起。高优先级文件（比如：user 目录）会覆盖低优先级文件。

读取文件时会区分大小写（例外：read…IIgnoreCase）。

读取配置数据

读取配置数据时需要生成 SIHmiSettingsQt 类的一个实例。方法 **readConfigEntry** 或 **readConfigEntries** 可用于读取配置文件中的一条或几条条目。

另外，方法 **readConfigEntryIgnoreCase** 和 **readConfigEntriesIgnoreCase** 可用于从 INI 文件中读取数据，此时不区分
段落/条目的大小写。如果将其中某个方法应用到其他格式的配置文件中（比如 XML 格式），会报告错误。

配置文件的名称是第一个参数的第一个部分，无需指定后缀名。

示例：

```
#include "slhmisettingsqt.h"

void SlBspClass::bspFunction(...)
{
    SlHmiSettingsQt settingsAccess;
    QVariant varResult;
    long nError = 0;

    nError = settingsAccess.readConfigEntry(
                "mysettings/MySection", // 文件&段落
                "MyEntry",              // 条目
                varResult);             // 结果

    if(SL_SUCCEEDED(nError))
    {
        ...
    }
}
```

在本例中要读取配置文件“mysettings”（全名为 mysettings.ini）中段落“MySection”中的条目“MyEntry”。

GUIFrameWork\SlExGuiSettings 的示例目录下有一个示例。

写入配置数据

写入配置数据时需要生成 SlHmiSettingsQt 类的一个实例。方法 **writeEntry** 或 **writeEntries** 可用于向配置文件写入一条或几条条目。

配置文件的名称是第一个参数的第一个部分，无需指定后缀名。

```
#include "slhmisettingsqt.h"

void SlBspClass::bspFunction(...)
{
    SlHmiSettingsQt settingsAccess;
    QVariant varValueToWrite;
    long nError = 0;

    varValueToWrite = 17;

    nError = settingsAccess.writeEntry(
"mysettings/MySection", // 文件&段落
                "MyEntry",               // 条目
                varValueToWrite);        // 值

    if(SL SUCCEEDED(nError))
    {
        ...
    }
}
```

在写配置文件时，数据始终保存在最高优先级的目录中，也就是“/user/sinumerik/hmi/cfg”，以便通过删除 user 目录下的数据来恢复出厂设置。如果您不希望如此，而是希望改写特定目录中的配置文件，必须在创建 SlHmiSettingsQ 类一同指定该目录的路径，比如：

```
SlHmiSettingsQt settingsAccess("/card/oem/sinumerik/hmi/cfg");
```

在写配置数据时 XML 格式是缺省格式。 如果您希望使用另一种格式（CFG 或 INI），必须通过 **defineDefaultFormat** 设置该格式：

```
SlHmiSettingsQt settingsAccess;
settingsAccess.defineDefaultFormat("ini");
```

GUIFrameWork\SlExGuiSettings 的示例目录下有一个示例。

列明配置数据

列明配置数据时需要生成 SlHmiSettingsQt 类的一个实例。方法 **listSections** 或 **listEntries** 可用于列明配置文件的内容及其段落。

配置文件的名称是第一个参数的第一个部分，无需指定后缀名。

```
#include "slhmisettingsqt.h"

void SlBspClass::bspFunction(...)
{
  SlHmiSettingsQt settingsAccess;
  QStringList strSectionList;
  long nError = 0;

  nError = settingsAccess.listSections(
             "regie",
             strSectionList);

  if(SL_SUCCEEDED(nError))
  {
    for(long nIndex = 0; nIndex < strSectionList.count(); nIndex++)
    {
      QStringList strEntriesList;
      QString szSection = "regie/";

      szSection += strSectionList[nIndex];

      nError = settingsAccess.listEntries(
                 szSection,
                 strEntriesList);

      for(long nIndex2 = 0; nIndex2 < strEntriesList.count(); nIndex2++)
      {
        ...
      }
    }
  }
}
```

INI 文件格式

Windows 系统下有效的 INI 格式是配置文件的缺省格式。

INI 文件包含的段落的名称在方括号中给出。每个段落中都包含了由“关键字-值”对构成的条目。关键字用等号赋值。段落外不能有条目。

注释用分号“；”或井号“#”开头。

示例：

```
# 注释行
; 另一条注释行

[My Section]
My_Entry = My_Value
```

该 INI 文件的开头是两条注释。 接着是名为“My_Section”的段落，该段落包含了关键字“My_Entry”。 关键字的值为“My_Value”。

XML 文件格式

XML 文件始终以 XML 定义开头：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

XML 配置文件的根标签始终是<CONFIGURATION>。

段落是<CONFIGURATION>的子标签，段落也可以包含子段落。因此它的分级要比 INI 格式多得多。

每个条目作为最小的标签写入，也就是说，数值以属性的形式指定。有以下两种属性：

- **value:** 文本格式的值。
- **type:** 类型数据，该文本必须为 QVariant 所知（参见 Qt 文档中关于 QVariant::nameToType()的说明）

注释以<!--开头，以 -->结尾，可以占据几行。

该文件最好以 UTF-8 代码保存，以便特殊字符能作为数值保存。

示例：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<!-- 注释行 -->

<CONFIGURATION>
    <My_Section>
                <My Entry value="My Value" type="QString" />
    </My_Section>
</CONFIGURATION>
```

本例是上文 INI 文件对应的 XML 格式。 其中也有一个含条目“My-Entry”的段落“My_Section”。 该条目值为“My_Value”。

注

这只是一个类似于 XML 的格式，其嵌套深度不超过 INI 文件的深度。在 `SlHmiSettingsQt` 类中不允许使用 XPath。

CFG 文件格式

配置文件的 CFG 格式是一个二进制格式。INI 格式和 XML 格式可以通过一个转换程序转换为 CFG 格式，以便在程序运行时可以更快地被读取。

INI 格式或 XML 格式借助命令行转换程序转换为 CFG 格式时需要使用以下命令行：

```
slhmiconvertercmd -in <file>.xml -out <file>.cfg -converter
slhmisettingsconverter.SlHmiSettingsConverter
```

转换程序也可以将 CFG 格式转换为 INI 格式或 XML 格式。

除了命令行转换程序外，还有一个采用图形化用户界面的转换程序。该程序名为 slhmiconvertergui。

4.17 将 HMI 对话框集成到 SINUMERIK Operate 中

和 HMI 标准对话框一样，HMI OA 对话框通常也是通过
操作区域切换条内的软键选中的。为此所必需的设置将在本章中加以介绍。

HMI 由名为系统管理器的应用程序启动和管理。在运行时，系统管理器启动主机进程，随后将 HMI 对话框载入其中。

可通过文件“systemconfiguration.ini”配置系统管理器。

系统配置

可在配置文件“systemconfiguration.ini”中确定 HMI 操作区域的组成结构和系统的 HMI 对话框。

若要将一个 HMI 对话框集成到系统中，则须将文件“systemconfiguration.ini”保存到以下其中一个目录下：

- **<安装路径>/user/sinumerik/hmi/cfg**
- **<安装路径>/oem/sinumerik/hmi/cfg**

该配置文件由几个段落组成：

表 4-215：“systemconfiguration.ini”中的段落

段	含义
miscellaneous	初始操作区的各种设置。
processes	由系统管理器启动的进程：比如载入 HMI 对话框的 HMI 主机进程。
areas	操作区域的配置。
dialogs	HMI 对话框的配置。

[miscellaneous]段
可在本段中完成多种设置。但一般只在此处修改初始操作区域。

表 4-216：[miscellaneous]段的条目

关键字	值
startuparea	初始操作区域的名称
strictStartup	只有成功载入所有组件后，HMI 才启动。 数值：true 或 false
HMIHasNotNCK	HMI 没有连接到 NCK。 数值：true 或 false
HMIHasNotPLC	HMI 没有连接到 PLC。 数值：true 或 false
doubleAreaMenuTime	双击操作区域按键以返回前一操作区域的最长时间，单位：毫秒。

示例：

```
startuparea = MyArea
```

[processes]段
在该段中指定由系统管理器启动的进程。 可以启动后台应用程序或启动用于自定义 HMI OA 对话框的额外的主机进程。可以为进程设置以下值：

表 4-217：进程配置值

值	含义
image	可执行文件的名称，比如：HMI 主机进程的 slsmhmihost
process	进程的符号名称
cmdline	命令行，启动时会传递给进程
background	如果是后台应用程序，将其设为“true”
strictStartup	“false”➔ 在启动时忽略该进程中的错误。 “true” ➔ 该进程中的错误会导致 HMI 关机。 （该值会改写“miscellaneous”段中的同名设置）

本例中配置了一个名为 SimpleEXE.exe 的应用程序，在系统启动时启动该程序：

```
PROC500= image:=SimpleEXE, process:=SimpleOEM, background:=true
```

[areas]段
此段用于配置 HMI 的操作区域。

可为一个操作区域设置以下值：

表 4-218：操作区域配置值

值	含义
name	操作区域的符号名称
dialog	本操作区域的初始对话框
panel	本操作区域中显示的面板，比如：标题

示例：

```
AREA500= name:=MyArea, dialog:=MyDialog, panel:=SlHdStdHeaderPanel
```

本例中配置了一个操作区域，该区域的初始对话框是“MyDialog”并显示标准 HMI 标题。

[dialogs]段
此段用于配置 HMI 的对话框。
可以为对话框设置以下值：

表 4-219：对话框配置值

值	含义
name	对话框的名称。该名称会成为对话框类对象的名称。
implementation	实现对话框的库和类，格式为 *bibliothek.klasse*。 没有实现任何自定义的对话框类时，在此处指定 *slgfw.SlGfwHmiDialog*。
process	装载对话框所属库的进程。此处指定的是进程配置中的符号名。
preload	指定在 HMI 启动时是否直接实例化对话框。 数值：true 或 false
terminate	指定是否在隐藏某对话框后摧毁该对话框。 数值：true 或 false

值	含义
cmdline	命令行，在初始化(init())和显示对话框(open())时传递给对话框。 当经过转换的对话框配置文件的名称和**对话框对象**不同（小写）时，必须在该命令行中指定该配置文件。 示例： cmdline:=“-conf mydialog.hmi“

示例：

```
DLG500= name:=MyDialogInstance,
implementation:=mydialog.MyDialog, process:=SlHmiHost1,
preload:=true, cmdline:="-conf mydialog.hmi"
```

本例配置了一个在运行时获得“MyDialogInstance”名称的对话框。其中会从 mydialog.dll 库（或 libmydialog.so）中的 MyDialog 类生成一个实例。 Dll 会装载到 HMI 主机进程“SlHmiHost1”，即标准的 HMI 主机进程。该对话框在启动时直接实例化，从文件 mydialog.hmi 中载入配置。

标准 HMI 主机进程中对话框的完整配置示例：

```
[areas]
AREA500= name:=MyArea, dialog:=MyDialog, panel:=SlHdStdHeaderPanel

[dialogs]
DLG500= name:=MyDialogInstance, implementation:=mydialog.MyDialog,
process:=SlHmiHost1, preload:=true, cmdline:="-conf mydialog.hmi"
```

自定义 HMI 主机进程中对话框的完整配置示例：

```
[processes]
PROC500= image:=slsmhmihost, process:=MyHmiHost, cmdline:="-
ORBCollocationStrategy direct"

[areas]
AREA500= name:=MyArea, dialog:=MyDialog, panel:=SlHdStdHeaderPanel

[dialogs]
DLG500= name:=MyDialogInstance, implementation:=mydialog.MyDialog,
process:=MyHmiHost, preload:=true, cmdline:="-conf mydialog.hmi"
```

条目<empty>
在“processes”、“areas”和“dialogs”段中，可以通过<empty> 条目再次删除完成的配置。

下例要隐藏标准操作区域“参数”(AreaParameter, SlParameter)：

```
[areas]
AREA001 = <empty>

[dialogs]
DLG002 = <empty>
```

! **重要提示**

在“processes”、“areas”和“dialogs”段中，编号范围
500-999 预留给 OEM 用户。若使用的编号小于 500，则可能会覆盖西门子的基本组件。

操作区域菜单的配置

操作区域菜单用于切换在“systemconfiguration.ini”中配置的操作区域。对于每个已配置的操作区域，在水平软键栏中均有一个软键，可通过此软键选择相应的区域。

操作区域菜单在操作区域软键上以文本形式显示在“systemconfiguration.ini”中配置的操作区域的名称。此时会自动为每个操作区域在水平软键栏上找到一个空闲的软键。如需在软键上用外语和图标来显示操作区域，则还需要进一步的配置。

该配置在配置文件“slamconfig.xml”中进行。该文件和“systemconfiguration.ini”位于同一目录下。

可在配置文件中为已在文件“systemconfiguration.ini”中配置过的每个操作区域创建一个段。此段的名称必须为已配置的操作区域的名称，比如 MyArea。

可为一个操作区域设置以下值：

表 4-220：slamconfig.xml 中用于操作区域配置的值

值	含义
TextId	外语文本的文本 ID，作为软键标记显示。
TextContext	外语文本的上下文。
TextFile	上下文和外语文本所属的文本文件的名称。
Picture	用作软键图标的图片文件的名称。
SoftkeyPosition	区域软键的固定软键位置。此时位置 1 到 8 为第一 ETC 级，9 到 16 为第二 ETC 级，以此类推。
AccessLevel	显示软键需要的最低访问级别。没有给定该值时，则会将访问级别设为 7（= 钥匙开关位置 0）。

示例：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<CONFIGURATION>
    <MyArea>
        <TextId value="MY AREA" type="QString"/>
        <TextFile value="mytextfile" type="QString"/>
        <TextContext value="mycontext" type="QString"/>
        <Picture value="mypicture.png" type="QString"/>
        <SoftkeyPosition value="7" type="uint"/>
        <AccessLevel value="5" type="uint" />
    </MyArea>
</CONFIGURATION>
```

在该配置文件中配置了操作区域“MyArea”的软键。 在运行时，该软键位于位置 7 上并从访问级别“钥匙开关位置 2”起进行显示。 软键所显示的文本保存在文本文件“mytextfile_xxx.ts”中，在上下文“mycontext”下，其文本 ID 为“MY_AREA”。显示 mypicture.png 作为图标。

! **重要提示**

操作区域位置 7 已预留给 OEM 客户。

4.18 将 OEMFrame 应用程序集成到 SINUMERIK Operate 中

通常情况下 OEMFrame 应用程序可以通过区域切换栏的软键选中。为此所必需的设置将在本章中加以介绍。

HMI 是由一个名为“系统管理器”的应用程序启动和控制的。此系统管理器也可控制 OEMFrame 应用程序。

可通过文件“systemconfiguration.ini”配置系统管理器。

系统配置

可在配置文件“systemconfiguration.ini”中确定 HMI 操作区域的组成结构。

若要将一个 OEMFrame 应用程序集成到系统中，则须将文件“systemconfiguration.ini”保存到以下其中一个目录下：

- **<安装路径>/user/sinumerik/hmi/cfg**
- **<安装路径>/oem/sinumerik/hmi/cfg**

下面将介绍“systemconfiguration.ini”文件中与 OEMFrame 应用程序的集成相关联的部分。

[processes]段
在该段中声明由系统管理器进行管理的进程。在此处还应对要集成的 OEMFrame 应用程序进行说明。以下各项与 OEMFrame 应用程序相关：

表 4-221：[processes]段中与 OEMFrame 应用程序相关的条目

值	含义
process	OEMFrame 应用程序的符号名称。设计操作区域时会用到。
cmdline	命令行，启动时会传递给“oemframe.exe”进程。 必须使用“\\”作为路径分隔。
oemframe	对于 OEMFrame 应用程序，始终设置为“true”。
windowname	OEMFrame 应用程序的窗口名称，可使用随附的 Findwindow 工具确定开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → 工具 →Find Window）
classname	OEMFrame 应用程序的 Class 名称，可使用随附的 Findwindow 确定（开始菜单项参见“windowname”）
deferred	true:OEMFrame 应用程序不是在 HMI 启动时启动，而是在首次选择后才启动。
startupTime	只在“deferred:=false”时相关。 相关进程按以下方式启动： immediately → 立即（默认） afterServices → 在所有服务都引导完成后。 afterGuis → 在所有 GUI 组件都引导完成后。

值	含义
gimmekeys	由 OEMFrame 应用程序操控的系统配置按键的使能掩码。参数设置以位掩码(bitmask)的形式进行。 （参见示例 2）
disablekeys	键盘过滤器属性的设置。参数设置以位掩码(bitmask)的形式进行。 （参见示例 2）
disablekeyshigh	快捷键的映射功能。参数设置以位掩码(bitmask)的形式进行。 （参见示例 4）
menuselectkey	用于修改操作区域菜单要显示的按键（默认：F10）。此时的值为组合键 Shift、Alt、Ctrl 与虚拟按键代码（同 Microsoft 的定义）的或(OR)组合。 （参见示例 3）

gimmekeys 位掩码可通过以下方式设置：

表 4-222：gimmekey 位掩码

位	按键	含义
0	F1 – F8	水平软键（上边栏，HU）
1	Shift+F1 – Shift+F8	垂直软键（右边栏，VR）
2	Ctrl+F1 – Ctrl+F8	水平软键（下边栏，HL）
3	Shift+Ctrl+F1 – Shift+Ctrl+F8	垂直软键（左边栏，VL）
4	F9	回调
5	Shift+F9	ETC 切换
6	F10	操作区域菜单
7	Shift+F10	M 键
8	F11	通道切换键
9	Shift+F11	M 键（硬键）
10	F12	信息/帮助
11	Shift+F12	自定义键（硬键）
12	ESC	报警取消
13	HOME	窗口切换键
14	END	PROGRAM（硬键）
15	PAGE UP	ALARM（硬键）
16	PAGE DOWN	TOOL OFFSET（硬键）
17	HOME（数字区）	PROGRAM MANAGER（硬键）
18	F13 – F20	扩展水平软键（上边栏，HU）
19	Shift+F13 – Shift+F20	扩展垂直软键（右边栏，VR）和右侧直接按键 HT8
20	Ctrl+F13 – Ctrl+F20	扩展水平软键（下边栏，HL）
21	Shift+Ctrl+F13 – Shift+Ctrl+F20	扩展垂直软键（左边栏，VL）和左侧直接按键 HT8

OEMFrame 应用程序的 gimmekey 位掩码默认设为 0xF，即所有 F1-F8 的组合键都提供给了 OEMFrame 应用程序。通过设置其他的位，相应的键（组合键）可由 OEMFrame 应用程序本身来操控。否则系统配置将接管求值计算并且 OEMFrame 应用程序会获得完全未被提供的键（组合键）。

disablekeys 位掩码可通过以下方式设置：

表 4-223：disablekeys 位掩码

位	按键	含义
0-7	预留	
8	(Shift)+Ctrl+F1	底边及左侧软键栏（HL，VL）
9	(Shift)+Ctrl+F2	底边及左侧软键栏（HL，VL）
10	(Shift)+Ctrl+F3	底边及左侧软键栏（HL，VL）
11	(Shift)+Ctrl+F4	底边及左侧软键栏（HL，VL）
12	(Shift)+Ctrl+F5	底边及左侧软键栏（HL，VL）
13	(Shift)+Ctrl+F6	底边及左侧软键栏（HL，VL）
14	(Shift)+Ctrl+F7	底边及左侧软键栏（HL，VL）
15	(Shift)+Ctrl+F8	底边及左侧软键栏（HL，VL）
16	预留	
17	预留	

OEMFrame 应用程序的 disablekeys 位掩码默认设为 0x3FFFF，即所有的按键组合都被过滤了出去，因此不会提供给 OEMFrame 应用程序。如果将某个位设为 0，则会取消相应按键组合的键盘过滤器并且 OEMFrame 应用程序会接收该设置。如果 OEMFrame 应用程序需要接收底边及左侧软键条上的全部软键，则将 disablekeys 位掩码设为 0x300FF。

disablekeyshigh 位掩码可通过以下方式设置：

表 4-224：disablekeyshigh 位掩码

位	含义
0-28	预留
29	快捷键 CTRL-F1 到 CTRL-F8 映射给快捷键 CTRL-F13 到 CTRL-F20。
30-31	预留

快捷键的映射可能必须进行，因为在某些情况下操作系统已经对 CTRL-F4 和 CTRL-F6 作出响应。

gimmekeys 位掩码、disablekeys 位掩码和 disablekeyshigh 位掩码既可以使用十进制（如 31）也可以使用十六进制（如 0x1F）。

示例 1：

```
[processes]
PROC500=process:=notepadOEM, cmdline:="C:\\WINDOWS\\system32\\notepad.exe",
oemframe:=true, deferred:=true, windowname:="Untitled - Notepad",
classname:="Notepad"

PROC501=process:=calcOEM, cmdline:="C:\\WINDOWS\\system32\\calc.exe",
oemframe:=true, deferred:=true, windowname:="Calculator",
classname:="SciCalc"

[areas]
AREA500=name:=AreaNote, process:=notepadOEM
AREA501=name:=AreaCalc, process:=calcOEM
```

在该示例中将两个 Windows 应用程序“notepad.exe”和“calc.exe”配置为 OEMFrame 应用程序。

示例 2：

```
[processes]
PROC500= process:=keycatcherOEM, cmdline:="keycatcher.exe", oemframe:=true,
deferred:=true, windowname:="keycatcher", classname:="QWidget",
gimmekeys:=0x1F, disablekeys:=0x300FF

[areas]
AREA500=name:=AreaKeyCatcher, process:= keycatcherOEM
```

在该示例中将 Windows 应用程序“keycatcher.exe”集成了进来。此时将全部 4 个软键条和回调键都提供给了 Windows 应用程序。底边和左侧软键条的键盘过滤器被取消。

示例 3：

```
[processes]
PROC500= process:=keycatcherOEM, cmdline:="keycatcher.exe", oemframe:=true,
deferred:=true, windowname:="keycatcher", classname:="QWidget",
gimmekeys:=0x4F, disablekeys:=0x300FF, menuselectkey:=Key_Control|0x7B

[areas]
AREA500=name:=AreaKeyCatcher, process:= keycatcherOEM
```

在该示例中将 Windows 应用程序“keycatcher.exe”集成了进来。此时将全部 4 个软键条和 F10 键都提供给了 Windows 应用程序。欲在 Windows 应用程序中显示操作区域菜单（F10 不再由系统配置处理），可按下 Ctrl+F12。

示例 4：

```
[processes]
PROC500= process:=keycatcherOEM, cmdline:="keycatcher.exe", oemframe:=true,
deferred:=true, windowname:="keycatcher", classname:="QWidget",
gimmekeys:=0xF, disablekeys:=0x300FF, disablekeyshigh:=0x20000000

[areas]
AREA500=name:=AreaKeyCatcher, process:= keycatcherOEM
```

在该示例中将 Windows 应用程序“keycatcher.exe”集成了进来。此时将全部 4 个软键条都提供给了 Windows 应用程序。快捷键 CTRL-F1 到 CTRL-F8 映射给快捷键 CTRL-F13 到 CTRL-F20。

[areas]段

在此部分配置 HMI 的操作区域。

以下各项与 OEMFrame 应用程序相关：

表 4-225：[areas]段中与 OEMFrame 应用程序相关的条目

值	含义
name	操作区域的符号名称
process	[processes]段中 OEMFrame 应用程序的名称
panel	要使用的面板(Header)的名称。 当前只有“SlHdStdHeaderPanel”可用于 OEM 应用程序。

示例：

```
[areas]
AREA600= name:=AreaOEM, process:=notepadOEM
AREA601= name:=AreaCalc, process:=calcOEM, panel:=SlHdStdHeaderPanel
```

! **重要提示**

在“processes”和“areas”段中编号范围
500-999 预留给 OEM 用户。若使用的编号小于 500，则可能会覆盖西门子的基本组件。

! **重要提示**

不支持使用 HMI 高级版的编程接口的 OEMFrame 应用程序。

! **重要提示**

为避免写入错误，“[process]段”和“[areas]段”中的条目只能使用随附工具“FindWindow”（开始 → 程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create MyHMI -3GL → 工具 → Find Window）确定。

更多信息参见 SINUMERIK Operate 编程包中的文档“FindWindow.pdf”。

[miscellaneous]段
可在本段中完成多种设置。但一般只在此处修改初始操作区域。

表 4-226：[miscellaneous]段的条目

关键字	值
startuparea	初始操作区域的名称

示例：

```
[miscellaneous]
startuparea = AreaOEM
```

操作区域菜单的配置

操作区域菜单用于切换在“systemconfiguration.ini”中配置的操作区域。对于每个已配置的操作区域，在水平软键栏中均有一个软键，可通过此软键选择相应的区域。

操作区域菜单在操作区域软键上以文本形式显示在“systemconfiguration.ini”中配置的操作区域的名称。此时会自动为每个操作区域在水平软键栏上找到一个空闲的软键。如要将操作区域分配给操作区域菜单中某个特定的软键或是在软键上显示外语文本或操作区域图标，则还需进行其他的配置。

该配置在配置文件“slamconfig.xml”中进行。该文件和“systemconfiguration.ini”位于同一目录下。

可在配置文件中为已在文件“systemconfiguration.ini”中配置过的每个操作区域创建一个段。此段的名称必须为已配置的操作区域的名称，比如 AreaOEM。
可为一个操作区域设置以下值：

表 4-227：slamconfig.ini 中用于操作区域配置的值

值	含义
TextId	外语文本的文本 ID，作为软键标记显示。
TextContext	外语文本的上下文。
TextFile	上下文和外语文本所属的文本文件的名称。
Picture	用作软键图标的图片文件的名称。
SoftkeyPosition	区域软键的固定软键位置。此时位置 1 到 8 为第一 ETC 级，9 到 16 为第二 ETC 级，以此类推。
AccessLevel	显示软键需要的最低访问级别。没有给定该值时，则会将访问级别设为 7（= 钥匙开关位置 0）。

示例：

```
[AreaOEM]
; Text-ID of a language dependent text
TextId = MY_AREA
; File name of the text file which contains the Text-ID
TextFile = mytextfile
; Context in the text file to which the Text-ID is assigned to
TextContext = mycontext
; File name of an icon shown on the area softkey
Picture = mypicture.png
; Position of the area softkey on area menu,
; If no position is specified, an empty position is searched
SoftkeyPosition = 7
; Access level of the area softkey
AccessLevel = 5
```

在该示例中配置了操作区域“AreaOEM”的区域软键。 该软键位于操作区域菜单的位置 7 上并从访问级别“钥匙开关位置 2”起进行显示。 软键所显示的文本保存在文本文件“mytextfile_xxx.ts”中，在上下文“mycontext”下，其文本 ID 为“MY_AREA”。显示 mypicture.png 作为图标。

! **重要提示**

操作区域位置 7 已预留给 OEM 客户。

注

一个 OEMFrame 应用程序由多个窗体组成时，最好将这些窗体整合到一个 MDI 框架下，以确保系统管理器正确执行区域切换。

4.19 设置 OEMFrame 应用程序(oemframe.ini)

通过文件“oemframe.ini”可继续对 OEMFrame 应用程序进行参数设置。

会为每个要存放参数的 OEMFrame 应用程序都创建一个独立的段，其名称应与相应的程序文件的名称一致（不含文件扩展名）。

“oemframe.ini”保存在目录“..\compat\oem”下。 如还未存在文件“oemframe.ini”，请在此处创建。

参数一览

表 4-228： “oemframe.ini”中的参数

条目	含义	缺省
WindowsStyle_On	表示要分配给窗口的属性。	0
WindowsStyte_Off	表示窗口所不具有的属性。	0
x	OEMFrame 应用程序的水平起始坐标（单位：像素）	0
y	OEMFrame 应用程序的垂直起始坐标（单位：像素）	0
Width	OEMFrame 应用程序的宽度（单位：像素）	桌面(Desktop)宽度
Height	OEMFrame 应用程序的高度（单位：像素）	桌面(Desktop)高度
nDelayInitComplete	延迟向系统管理器反馈。	0
fSearchOnlyForTask-Window	说明在“systemconfiguration.ini”中指定的窗口是否属于同样在此处指定的任务。	1
fRestoreTaskWindow	确定退出从 OEMFrame 应用程序中调用的应用程序时的特性。	0
fKeepPlacement	取消大小调整。	0
fForceTaskFocus fSearchForPopUps	该参数确定启动 OEMFrame 应用程序时显示哪个窗口。	0 1
nInitShowMode	启动 OEMFrame 应用程序时窗口显示的状态。	SW_SHOWMIN-NOACTIVE
nShowMode	OEMFrame 应用程序的窗口显示状态。	SW_SHOW-NORMAL
nUnShowMode	OEMFrame 应用程序的窗口隐藏状态。	SW_SHOWMIN-NOACTIVE
fWinForms	在涉及“Windows Forms Application”时进行设置。	0

WindowsStyle_On / WindowsStyle_Off

Windows 应用程序的外观主要借助 Windows API 函数 SetWindowLong 来确定。调用函数 SetWindowLong 时，应用程序的外观通过长度为 8 个字节的字来控制。使用属性 WindowStyle_On 和 WindowStyle_Off 可修改其中两个字节。在下表中加以说明：

表 4-229：使用 WindowStyle 属性的控制方式

0000	0000	xxxx	xxxx	0000	0000	0000	0000
		1010		标题			
		1000		边界			
		0100		对话框的窗口边框样式			
		0010		垂直滚动条			
		0001		水平滚动条			
			1000	系统菜单			
			0100	边框(Thickframe)			
			0010	Minimize-Box（最小化按钮）			
			0001	Maximize-Box（最大化按钮）			

此处以二进制显示的特征值会以十进制数的形式赋给 WindowStyle 属性。二进制与十进制之间的相互换算可使用 WINDOWS 的计算器进行，其通常在“附件”应用程序组中。

示例：
必须对系统菜单以及水平和垂直滚动条的属性进行定义。根据表格应为：

```
0000 0000 0011 1000 0000 0000 0000 0000 二进制或
0038 0000 Hex.
现在调用计算器并且
S 点击选择按钮“Hex”
S 输入数字 380000（可以省略前面的零）
S 点击选择按钮“Dec”
S 通过菜单“Edit”复制结果 3670016
S 将结果添加到属性中
```

参数“WindowStyle_On”表示要分配给窗口的属性，参数“WindowStyle_Off”表示窗口所不具有的属性。

示例 1：
为编辑器 NOTEPAD 显示系统菜单以及水平和垂直滚动条：

```
[notepad]
WindowStyle_On = 3670016
```

示例 2：
不在编辑器 NOTEPAD 显示最小化按钮和最大化按钮：

```
[notepad]
WindowStyle_Off = 196608
```

X/Y

属性 X 和 Y 表示所集成的 Windows 应用程序的窗口的起始坐标，将屏幕的左上角作为测量原点。

X 是水平坐标，Y 是垂直向下的坐标。尺寸单位为像素(pixel)。

可用的工作区域取决于使用的屏幕布局。

Width

该属性表示 Windows 应用程序的窗口宽度，从窗口原点开始测量，根据属性 X，单位像素。

Height

该属性表示 Windows 应用程序的窗口高度，从窗口原点开始测量，根据属性 Y，单位像素。

nDelayInitComplete

一旦找到 Windows 应用程序的窗口，就会相应地向系统管理器发送一条消息。之后即可通过系统管理器来选择 Windows 应用程序。 使用参数“nDelayInitComplete”可将此类消息的发送延迟。如果 Windows 应用程序在生成其窗口后还必须执行需要较长时间的任务，并且如果系统管理器过早地激活窗口则会导致窗口无法正确显示，就需要使用此类延迟
。
（单位：毫秒，标准值：0）

示例：
一个 Windows 应用程序“app.exe”在生成其窗口后还要继续从数据库中读取状态数据，这些数据对于之后 Windows 应用程序的正确执行是非常必要的。Windows 应用程序的窗口在所有的状态数据都读取完毕后才可显示。该读取操作平均约持续 1 sec。需要进行以下参数设置：

```
[app]
;worst case
nDelayInitComplete = 2000
```

fSearchOnlyForTaskWindow

使用该参数指定在文件“systemconfiguration.ini”中通过 ClassName/WindowName 定义的窗口是否属于同样在此处指定的任务。 如果窗口属于指定的任务，则参数“fSearchOnlyForTaskWindow”值为 1。 如果窗口不属于指定的任务，则参数值“fSearchOnlyForTaskWindow”为 0。在查找指定的窗口时，不只局限于在“systemconfiguration.ini”中
所设置的任务的窗口，而是针对查找时存在于系统中的全部窗口。
（标志位 flag，默认值：1）

示例：
Windows 应用程序由多个进程组成，例如“startup.exe”和“user.exe”。 在文件“systemconfiguration.ini”中（只）记录了“startup.exe”，之后会从该进程中启动“user.exe”。 应用程序窗口属于“user.exe”，因此在单独查找“startup.exe”的窗口时不会被找到。需要进行以下参数设置：

```
[startup]
fSearchOnlyForTaskWindow = 0
```

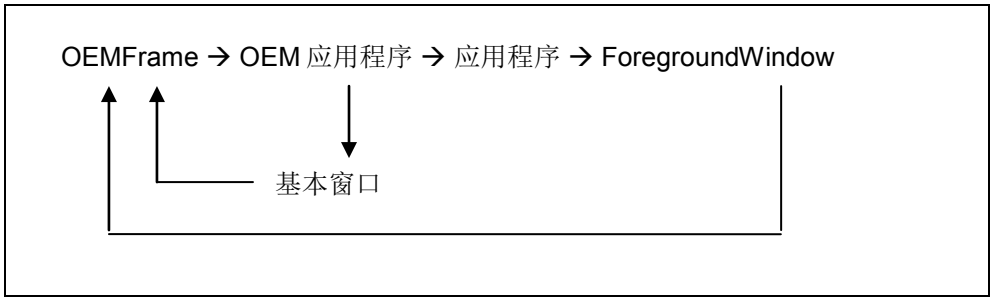
fRestoreTaskWindow

该参数确定退出从 OEMFrame 应用程序中调用的 Windows 应用程序时的特性（称为第 2 任务级）。

注

该情况无法完全掌控，因此应尽量避免！

一般在退出 OEMFrame 应用程序时，会对最后激活的窗口(ForegroundWindow)进行保存。在重新选择此 OEMFrame 应用程序时，会再次激活该窗口。如果从 OEMFrame 应用程序中又启动了其他的应用程序，激活的窗口一般属于 Windows 应用程序。



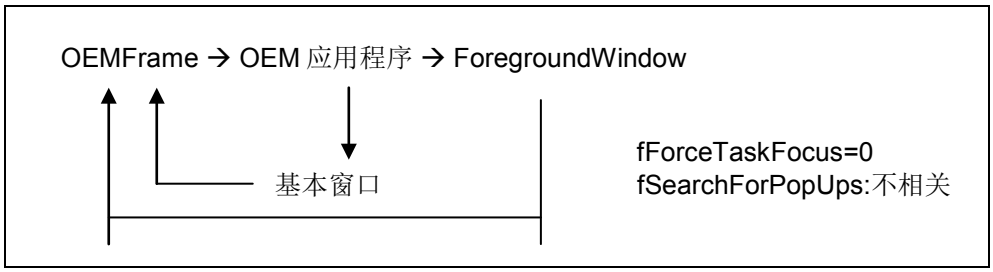
退出该 Windows 应用程序无法识别代理应用程序(OEMFRAME.EXE)。因此在这种情况下也无法将 OEMFrame 应用的窗口放至前台，有时还会导致在退出第二任务级时显示错误的场景(scenario)。

此时参数“fRestoreTaskWindow”是一种可靠的解决办法。如果该参数的值为 1，则在选择 OEMFrame 应用程序时或者选择从 OEMFrame 应用程序中启动的其他应用程序时始终会激活两个窗口。 作为第一个窗口，OEMFrame 应用程序的基本窗口会置于前台并且通过该窗口来设置“ForegroundWindow”。因此可确保，在退出第二任务级时始终显示 OEMFrame 应用程序的基本窗口。

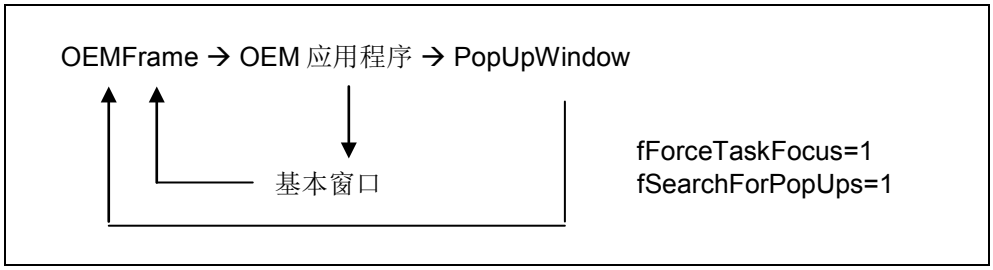
限制：
如果在启动第 2 个应用程序时在 OEMFrame 应用程序中激活了一个模态窗口(DialogBox)，则在退出第 2 个应用程序时该模态窗口会失去输入焦点。操作员此时必须将焦点重新置入（如使用鼠标）该对话框中。

fForceTaskFocus / fSearchForPopUps

使用这两个参数确定，OEMFrame 应用程序在退出和重新选择后激活哪个窗口。一般在区域切换时，会对最后激活的 OEMFrame 应用程序的窗口进行保存。该窗口在重新选择应用程序时会被再次激活。在 Windows API 中该窗口被称作 ForegroundWindow。



该（默认）特性适用于大多数的应用程序。但也有例外，可为其修改该特性。 如果这两个参数“fForceTaskFocus”和“fSearchForPopUps”设为 1，则在退出 OEMFrame 应用程序时不会查找 ForegroundWindow，而会查找应用程序基本窗口中激活的 PopUpWindow。如找到了 PopUpWindow，则在重新选择 OEMFrame 应用程序时会显示该窗口。如不存在 PopUpWindow，则在重新选择时会显示应用程序的基本窗口。



如果参数“fSearchForPopUps”的值设为
0，则不查找激活的 PopUpWindow。在退出并
重新选择应用程序时还会查找 OEMFrame 应用程序的基本窗口。 基本窗口通过 ClassName/WindowName 在文件“systemconfiguration.ini”中指定。

fKeepPlacement

该函数取消 OEM 应用程序基本窗口的大小调整（缩放）。通常应用程序在显示前会对其屏幕大小进行缩放。对不允许进行窗口缩放的应用程序进行大小调整时会引起显示问题。此时应关闭缩放。

示例：
应用程序“fixres.exe”应以所设置的窗口大小显示
。需要进行以下参数设置：

```
[fixres]
fKeepPlacement = 1
```

nInitShowMode / nShowMode / nUnShowMode

这三个参数确定在启动应用程序(nInitShowMode)时以及在显示和隐藏应用程序时以何种形式显示应用程序窗口。 参数“nShowMode”与显示（区域被激活）相关，参数“nUnShowMode 与隐藏相关。这两个参数的取值范围如下：

0: 应用程序窗口不可见(SW_HIDE)。

1: 应用程序窗口以原始形式（位置，大小）显示并获得输入焦点（SW_SHOWNORMAL，SW_NORMAL）。

2: 应用程序窗口被最小化并获得输入焦点(SW_SHOWMINIMIZED)。

3: 应用程序窗口被最大化(SW_SHOWMAXIMIZED)。

4: 显示应用程序窗口，但不获得输入焦点(SW_SHOWNOACTIVATE)。

5: 显示应用程序窗口并获得输入焦点(SW_SHOW)。

6: 应用程序窗口被最小化并失去输入焦点(SW_MINIMIZE)。

7: 应用程序窗口被最小化，但不获得输入焦点(SW_SHOWMINNOACTIVE)。

8: 显示应用程序窗口，但不获得输入焦点(SW_SHOWNA)。

9: 应用程序窗口以原始形式（位置，大小）显示(SW_RESTORE)。

10: 应用程序窗口以应用程序启动时的相同形式显示(SW_SHOWDEFAULT)。

注

默认设置适合于大多数的使用情况。但在某些使用Borland-Delphi开发的应用程序中，默认设置可能会导致显示问题（窗口移动等问题）。此时的解决办法是设置参数
nUnShowMode = 0以及
fKeepPlacement = 1。

---

fWinForms

如果 OEMFrame 应用程序涉及“Windows Forms Application”，则必须进行以下的参数设置：

```
[app-name]
fWinForms = 1
```

注

如果不设置参数，则 OEMFrame 应用程序不会以最大化窗口启动或者大小调整（X，Y，Width，Height）无效。

4.20 OEM 子目录及版本识别

OEM 子目录

如果集成了多个 OEM 应用程序（HMI 对话框或 OEMFrame 应用程序），则可将其分别分配到各自的 OEM 子目录中。这样做的好处是可将文件与 OEM 应用程序清晰的分类。 这尤其适用于文件“systemconfiguration.ini”，因为在复制时总是存在将其误覆盖并因此中断其他 OEM 应用程序的风险。

OEM 子目录可使用任意目录名称在“/oem/sinumerik/hmi”路径下进行创建。 该子目录在 cfg 目录（“/oem/sinumerik/hmi/cfg”）下的“systemconfiguration.ini”文件中进行声明。 在“oem_dirs”段中使用以下句法进行子目录的创建：

OEM_<数字>=<OEM 子目录>

示例：

```
[oem dirs]
OEM_1=oem_example
OEM_2=oem_new
```

在这两个 OEM 子目录（“/oem/sinumerik/hmi/oem_example”和“/oem/sinumerik/hmi/oem_new”）下可根据需要继续创建目录 appl、cfg、lng、hlp 等。

OEM 应用程序的版本识别

为使 OEM 应用程序具有自己的版本识别功能，可在“诊断”操作区域中使用软键“版本”进行查看，则需要创建相应的版本文件(versions.xml)。 这些文件应由 OEM 应用程序提供并保存在 OEM 目录（“/oem/sinumerik/hmi”）或相应的 OEM 子目录下（例如“/oem/sinumerik/hmi/oem_new”）。

版本文件的名称必须为“versions.xml”并以 XML 的类似格式创建。此处名称和版本最低应为：

示例（最低）：

```
<info>
    <Name>OEM1</Name>
    <Version>1.1</Version>
</info>
```

此外还可以定义内部版本及子组件或通过标签(Tag)<Path>指向其他版本文件。

示例（子组件，内部版本）：

```
<info>
    <Name>OEM1</Name>
    <Version>1.1</Version>
    <InternalVersion>999</InternalVersion>
    <Component>
        <Name>OEM1-ComponentX</Name>
        <Version>6.1</Version>
    </Component>
    <Component>
        <Name>OEM1-ComponentY</Name>
        <Version>00.00</Version>
        <InternalVersion>L00.00.06.00</InternalVersion>
    </Component>
</info>
```

示例(Path)：

```
<info>
    <Name>OEM1</Name>
    <Version>1.1</Version>
    <Component>
        <Name>OEM1-ComponentX</Name>
        <Version>6.1</Version>
        <Path>appl</Path>
    </Component>
</info>
```

在启动时对 OEM 目录以及可能存在的 OEM 子目录进行扫描并在目录“/oem”下生成上位版本文件。

注

只有当目录“/oem”下还不存在此类版本文件时，才会生成上位版本文件。意思是，在安装新的 OEM 应用程序时必须删除此类文件。

在此上位版本文件中记录 OEM 目录或 OEM 子目录中 OEM 应用程序的位置。如果同时使用 OEM 子目录和 OEM 目录，则应注意在版本文件中记录的内容不会被互相覆盖。

DLL 文件和 SO 文件的版本识别（仅限 HMI 对话框）

如需在显示版本时也直接从文件中读出 DLL 文件和 SO 文件的版本并显示这两个版本，以下 CPP 文件必须一同传送到 Visual Studio 项目中。此时可输入版本信息的位置会突出显示。没有该需要时可不传送 CPP 文件。

slexversion.cpp:

```
// version of dll/so-file
#define SL_VERSION "01.01.00.00"

#if defined(WIN32) || defined(WIN64)
#   define SL_VER_EXPORT __declspec(dllexport)
#else
#   define SL VER EXPORT
#endif

namespace SL_MODULE
{
    extern const char* const SL VER VERSION INFO = "@(#)$program version id:\n"
                             SL VERSION"\n";
}

extern "C"
{
    SL VER EXPORT const char* slVerGetVersion(void)
    {
        return SL MODULE::SL VER VERSION INFO;
    }
}
```

此外应在相应的 DLL/SO 文件所在的 appl 目录下保存“versions.xml”文件，该文件包含以下内容：

```
<info defaultFileType=" *.dll *.so"
      defaultFileVersion="01.01.00.00" LinkName="OEM">
    <Name>OEM1-ComponentX</Name>
    <Version>6.1</Version>
</info>
```

通过“defaultFileType”可直接从所有 DLL/SO 文件中读取版本信息，然后加以显示。 “defaultFileVersion”指定设定版本。如果设定版本和实际版本不一致，在版本显示时会标记一个明显的红色感叹号。

注

不会自动使用目录“appl”下的版本文件。 因此必须在上一级目录（如：<安装路径>/oem/sinumerik/hmi）中通过“Path”指向此文件：

```
<info>
    <Component>
        <Name>OEM1-ComponentX</Name>
        <Version>6.1</Version>
        <Path>appl</Path>
    </Component>
</info>
```

INI 文件的版本识别

为了同样能显示 INI 文件的版本，在 INI 文件的开头必须含有以下两行内容：

```
;VERSION:<Version> ;DATE:YYYY-MM-DD
;CHANGE:<Version> ;DATE:YYYY-MM-DD
```

示例：

```
;VERSION:06.02.08 ;DATE: 2002-01-07
;CHANGE:06.02.04 ;DATE: 2001-07-23
```

此外应在相应的 INI 文件所在的 cfg 目录下保存具有以下内容的“versions.xml”文件：

```
<info defaultFileType=" *.ini"
      defaultFileVersion="06.02.08" LinkName="OEM">
    <Name>ini-Dateien</Name>
</info>
```

通过“defaultFileType”的设置，会直接从文件中读取和显示所有 INI 文件中的版本信息。 “dafaultFileVersion”指定设定版本。如果设定版本和实际版本不一致，在版本显示时会标记一个明显的红色感叹号。

注

不会自动使用目录“cfg”下的版本文件。 因此必须在上一级目录（如：<安装路径>/oem/sinumerik/hmi）中通过“Path”指向此文件：

```
<info>
    <Component>
        <Name>OEM1-ComponentX</Name>
        <Version>6.1</Version>
        <Path>cfg</Path>
    </Component>
</info>
```

4.21 TRACE 输出

将 TRACE 的输出集成到软件中非常有助于诊断。一旦出现故障，即使是在释放版上也可以方便快速地开展诊断工作。

SINUMERIK Operate 提供全面的 TRACE 支持，其中管理了模块专有的 TRACE 位，这些位通常用于开/关 TRACE。由此产生的数据流会被缓存，由 TRACE 服务器传送到可选择的输出目标上。TRACE 位可在用户界面上显示并修改。

在第一次输出 TRACE 时，应用程序会先注册到“SolutionLine Trace Support”，以获得 TRACE 位并向用户界面传递文本。

方案

“SolutionLine Trace Support”编程接口采用以下方案：

TRACE 位，通常用于开/关特定主题的 TRACE。这些位可在程序中读写或用于程序中的其他控制。这些位也可仅用于显示在 TRACE 用户界面上。

TRACE 级，用于确定每个主题的 TRACE 涵盖的范围。“SolutionLine Trace Support”支持 8 个 TRACE 级。上级 TRACE 包含了下级 TRACE 的输出。

TRACE 筛选，用于筛选出特定进程或线程进行跟踪。

标准信息，包含源文件/行号、时间戳、进程 ID、线程 ID、模块名、主机名和栈指针等信息，可单独添加，使 TRACE 输出的信息更加丰富。

Datasink，即输出目标，用于选择 TRACE 数据流的输出通道。比如，可选择数据以文件输出。另一个输出通道是 TRACE 视图，它会保存 TRACE 的最后几行并显示在屏幕上。可以同时指定多个输出目标。

有**释放版 TRACE** 和**测试版 TRACE** 两个版本。在编译时激活了优化后，测试版从 release build 的程序代码中清除。

4.21.1 编程接口

添加库

使用 TRACE 输出时要在项目设置中检查库 `sltrc.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”。

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lsltrc
```

定义 TRACE 常数

在应用程序中编程 TRACE 输出前，首先要定义 TRACE 位和模块的名称。这些数据是 TRACE 的常数，必须在加入 TRACE 头文件前(sltrc.h)完成定义。

为使定义一目了然，您可以在项目中创建一份新的头文件，比如：mytrc.h，然后在其中定义这些常数。最后在文件末尾加入 TRACE 头文件 `sltrc.h`。

有以下几个常数：

SL_TRC_MODULE
该常数在任何情况下都要加以定义。它确定了模块的名称。该名称还可用于确定命名空间。

示例：

```
#define SL_TRC_MODULE MyOemApplication
```

SL_TRC_BIT<Nr>
一同有 28 个 TRACE 位，利用这些位您可以按“主题”划分 TRACE。比如：每个位用于开/关某一个类的 TRACE。每个位都包含了一个上述名称的常数。
SL_TRC_BIT0 代表 TRACE 位 0，SL_TRC_BIT27 代表 TRACE 位 27。在头文件中可以为这些常数指定简单易懂的名称。

示例：

```
#define TRC_BIT_MY_FORM   SL_TRC_BIT0
#define TRC_BIT_MY_SCREEN SL_TRC_BIT1
```

SL_TRC_BIT<Nr>_TEXT
可以为每个 TRACE 位定义一条标识文本，该文本会显示在 TRACE 服务中模块的位视图中。该文本要尽量简短。详细的说明可以在提示框中定义。

没有定义标识文本时，自动使用文本“BIT y”。 文本定义为“？”时，该位对应的复选框变为无效，文本为空。文本以“！”开头时，该位从 TRACE 服务的“Clear ...”和“Set ...”按钮中清除。文本以“%”开头时，位视图中该位对应的复选框变为无效，在操作界面上无法再修改该位。

示例：

```
#define SL_TRC_BIT0_TEXT "MyForm"
#define SL_TRC_BIT1_TEXT "MyScreen"
```

SL_TRC_BIT<Nr>_TOOLTIP
可以为每个 TRACE 位定义一条提示框文本。提示框包含了比标识文本更详细的说明。没有此项定义时，对应的 TRACE 位不显示提示框。

示例：

```
#define SL_TRC_BIT0_TOOLTIP "Trace for class MyForm"
#define SL_TRC_BIT1_TOOLTIP "Trace for class MyScreen"
```

SL_TRC_DEBUG
该预处理器常数确定了是否一同编译测试版的 TRACE 宏命令：SL_TRC_DEBUG = 1 表示一同编译，SL_TRC_DEBUG = 0 表示不一同编译。没有此项定义时，该常数根据是否存在预处理器常数_DEBUG 自动确定。

综合上述常数，一份自定义的 TRACE 头文件可能由以下内容组成：

```
#if !defined(MY_TRC_H)
#define MY_TRC_H

// 定义模块名称
#define SL_TRC_MODULE MyOemApplication

// 定义 TRACE 位
#define TRC_BIT_MY_FORM   SL_TRC_BIT0
#define SL_TRC_BIT0_TEXT "MyForm"
#define SL_TRC_BIT0_TOOLTIP "Trace for class MyForm"

#define TRC_BIT_MY_SCREEN SL_TRC_BIT1
#define SL_TRC_BIT1_TEXT "MyScreen"
#define SL_TRC_BIT1_TOOLTIP "Trace for class MyScreen"

// 加入 TRACE 机制
#include "sltrc.h"

#endif // MY_TRC_H
```

TRACE 宏命令

TRACE 机制的程序接口是一个宏命令接口，也就是说：要调用宏命令来编程 TRACE。

TRACE 使用以下宏命令：

SL_TRC（TRACE 级，TRACE 位，TRACE 输出）
设置了 TRACE 级和 TRACE 位后，输出 TRACE。此时多个 TRACE 位可以进行逻辑运算。

TRACE 输出可以如同 printf 命令中的一个字符串一样使用。

示例：

```
SL_TRC(SL_TRC_LEVEL1, TRC_BIT_MY_FORM, ("MyForm::MyForm() called. "));

int nError = initialize();
SL TRC(SL TRC LEVEL2, TRC BIT MY FORM,
("初始化错误:%d. ", nError));
```

SL_TRC_L<级号>_BIT<位号>(TRACE 输出)
设置了 TRACE 级和 TRACE 位后，输出 TRACE。此处针对的是释放版 TRACE 的输出。

TRACE 输出可以如同 printf 命令中的一个字符串一样使用。

示例：

```
// TRACE 级 1、位 0 的输出
SL TRC L1 BIT0("错误:0x%08x", nError);

// TRACE 级 3、位 1 的输出
SL TRC L3 BIT1("错误:0x%08x", nError);

// TRACE 级 1、任何位的输出
SL_TRC_L1_ANYBIT ("错误:0x%08x", nError);
```

SL_TRCD_L<级号>_BIT<位号>(TRACE 输出)
设置了 TRACE 级和 TRACE 位后，输出 TRACE。此处针对的是测试版 TRACE 的输出。

TRACE 输出可以如同 printf 命令中的一个字符串一样使用。

示例：

```
// TRACE 级 1、位 0 的输出
SL_TRCD_L1_BIT0("错误:0x%08x", nError);

// TRACE 级 3、位 1 的输出
SL_TRCD_L3_BIT1("Fehler:0x%08x", nError);

// TRACE 级 1、任何位的输出
SL_TRCD_L1_ANYBIT ("Fehler:0x%08x", nError);
```

SL_TRC_ERROR(TRACE 输出)
在 TRACE 服务中为模块 SlTrc 置位了位“Important Errors”后，输出 TRACE。在出现严重错误时所有模块应通过该宏命令输出故障信息。严重错误指会导致操作界面上出现明显异常的错误。

示例：

```
SL_TRC_ERROR ("ERROR:cant open file:%.100s",
SlTrc::QString2cz(strFileName));
```

重要提示

使用 SL_TRC 函数内的 QString 时，有两点要特别注意：

a) 要指定 QString 的最大长度，比如："%.100 s"，参见示例
b) 必须使用函数 SlTrc::QString2cz()，因为直接使用 QString 作为宏参数会引起嵌入式 Linux 系统崩溃。

重要提示

在“SL_TRC_L?_BIT?”中不允许有任何同样会生成 TRACE 输出的方法和函数。示例：

```
qint32 getCookie()
{
   SL_TRC_L8_ANYBIT("getCookieFunction");
}

void AnyClass::doSomething()
{
   QString szMsgText = "MsgText";
   QString timeStampText = "1.1.2009 15:00";

   SL TRC L8 ANYBIT("Timestring:%s, getCookie:%d, szMsgText:%s\n"
                                   ,SlTrc::QString2cz(timeStampText)
                                   ,getCookie()
                                   ,SlTrc::QString2cz(szMsgText));
}
```

解决方法 1：使用局部变量
在这种方法中，预先逐个调用所有方法，将返回值保存在局部变量中。为避免运行多余的代码，必须用宏命令SL_TRC_WILL_TRACE(级, 位)确定TRACE段。采用该方法后，上述例子的内容现在变为：

```
void AnyClass::doSomething()
{
   QString szMsgText = "MsgText";
   QString timeStampText = "1.1.2009 15:00";

   SL TRC WILL TRACE(SL TRC LEVEL8, SL TRC BIT5)
   {
      qint32 tmpCookie = getCookie();
      SL_TRC_L8_BIT5("Timestring:%s, getCookie:%d, szMsgText:%s\n"
                                    ,SlTrc::QString2cz(timeStampText)
                                    ,tmpCookie
                                    ,SlTrc::QString2cz(szMsgText));
   }
}
```

解决方法 2：用 toUtf8().constData()替代 QString2cz

采用该方法后，上述例子的内容现在变为：

```
void AnyClass::doSomething()
{
   QString szMsgText = "MsgText";
   QString timeStampText = "1.1.2009 15:00";

   SL_TRC_L8_ANYBIT("Timestring:%s, getCookie:%d, szMsgText:%s\n"
                                    ,timeStampText.toUtf8().constData()
                                    ,getCookie()
                                    ,szMsgText.toUtf8().constData());
}
```

必要的 TRACE 函数

SlTrc::QString2cz()
QString2cz 将一个 QString 转换成以 0 结束的 UTF-8 字符串。

```
const char * SlTrc::QString2cz( const QString& qstr );
```

该函数只能在 SlTrc-Trace 的参数中使用，因为它使用的是在输出 TRACE 时再次清空的内存。在 SL_TRC....语句外使用该函数肯定会造成内存泄漏（memory leak）。

示例：

```
QString str_1("String1");
QString str_2("String2");

SL TRC L1 BIT5("trace str 1:%.20s str 2:%.20s"
    , SlTrc::QString2cz(str_1), SlTrc::QString2cz(str_2));
```

有用的 TRACE 函数

SlTrc::ms()
函数“ms”返回毫秒级时间。

```
unsigned SlTrc::ms();
```

SlTrc::at()
函数“at”返回毫秒级时间文本。在必要时参数“Parameter _time”会返回毫秒级时间尺。

```
const char* SlTrc::at(unsigned *_time = 0);
```

SlTrc::ativ()
函数“ativ”返回毫秒级时间文本和持续时间。持续时间由传递参数确定。该参数必须用 SlTrc::ms()或 SlTrc::at(...)赋值。

```
const char* SlTrc::ativ(unsigned time);
```

示例：

```
void proc(int nArg)
{
    unsigned int tim = SL TRC TIME NOT SET;
    SL_TRC_L8_BIT0(">>> proc at:%.20s", SlTrc::at(&tim));
    ..
    SL_TRC_L8_BIT0("### proc at:%.20s", SlTrc::at());
    ..
    SL_TRC_L8_BIT0("<<< proc at:%.20s", SlTrc::ativ(tim));
}
```

提供以下输出：
>>> proc at: 12:23:37.455
proc at: 12:23:37.900
<<< proc at: 12:23:38.266(0.811)

4.21.2 生成 TRACE

启动 TRACE 服务，以生成 TRACE。在 TRACE 服务中设置需要跟踪的模块的 TRACE 位，并选择一个输出目标。随后可以启动应用程序，再次输出错误。

有多种方法可启动 TRACE 服务：

表 4-230： 启动 TRACE 服务的方法

	PCU50	Linux-Embedded	开发 PC
通过操作界面	X	X	X
通过开始菜单项	---	---	X
通过 SSH 客户端程序（如：Putty）	---	X	---

启动 TRACE 服务（通过操作界面）

在调试操作区下查找 TRACE 服务：
HMI ➔诊断➔ HMI-TRACE

启动 TRACE 服务（通过开始菜单项）

TRACE 服务通过以下开始菜单项来启动：

```
开始 →程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create
MyHMI -3GL →工具 → slTrcSvc
```

启动 TRACE 服务（通过 SSH 客户端程序）

准备工作

首先要打开防火墙中 TRACE 服务需要使用的端口，方可使用该服务。为此可通过诸如 WinSCP 之类的 SCP 客户端程序登录系统。

登录数据：
用户➔ ’manufact’
密码➔ ’SUNRISE’

在文件“/card/user/system/etc/basesys.ini”中修改以下条目：

```
[LinuxBase]
;FirewallOpenPorts=TCP/5900
```

在其中删除行首的注释符号，打开端口 5901：

```
[LinuxBase]
FirewallOpenPorts=TCP/5901
```

需要打开多个端口时，可以用空格隔开端口，比如：

```
[LinuxBase]
FirewallOpenPorts="TCP/5900 TCP/5901 TCP/102"
```

重启系统后，修改才生效。

启动 TRACE 服务

首先要通过“Putty”（可用的 SSH 和 Telnet 客户端程序）登录系统，以启动 TRACE 服务。登录数据和上文登录 SCP 户端程序一样。

启动 TRACE 服务：

```
sc openport tcp/5901
cd /siemens/sinumerik/hmi/autostart
./run_hmi -start sltrcsvc -qws -display VNC:1
```

TRACE 服务现在显示在 VNC 显示屏 1 上，也就是说，无法在相连的薄型客户端上看到该服务。

现在在开发计算机上您可以通过 VNC 客户端程序（如 UltraVNC）和 TRACE 服务连在一起。为此要指定系统的 IP 地址（VNC 服务器的地址）和显示号。比如 NCU 的 IP 地址为 192.168.1.1 时，VNC 服务器地址则为 192.168.1.1:1。VNC 客户端程序随后显示 TRACE 服务应用程序。

注

结束 SSH 客户端程序（如“Putty”）也会一同结束 TRACE 服务。

配置 TRACE 服务

在启动后 TRACE 服务显示 TRACE 位视图：

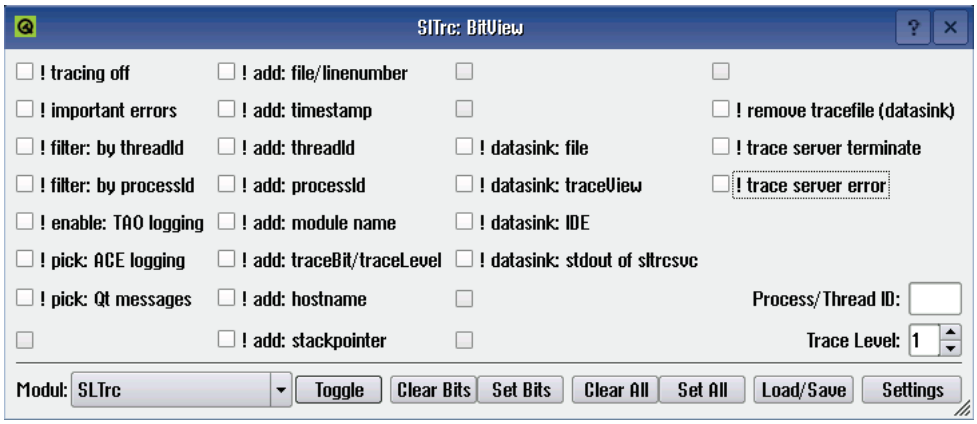


图 4-16:TRACE 服务的设置

首先在左下方的模块复选框中选择一个模块，激活该模块的 TRACE 输出。复选框中显示的是用常数 SL_TRC_MODULE 设置的模块名称。选中模块后便可以看到该模块的 TRACE 位。此处也可以看到为 TRACE 位指定的标识文本。

现在在该窗口中选择需要在 TRACE 数据流中输出的 TRACE 位。也可以在窗口的右下角设置该模块的 TRACE 级。

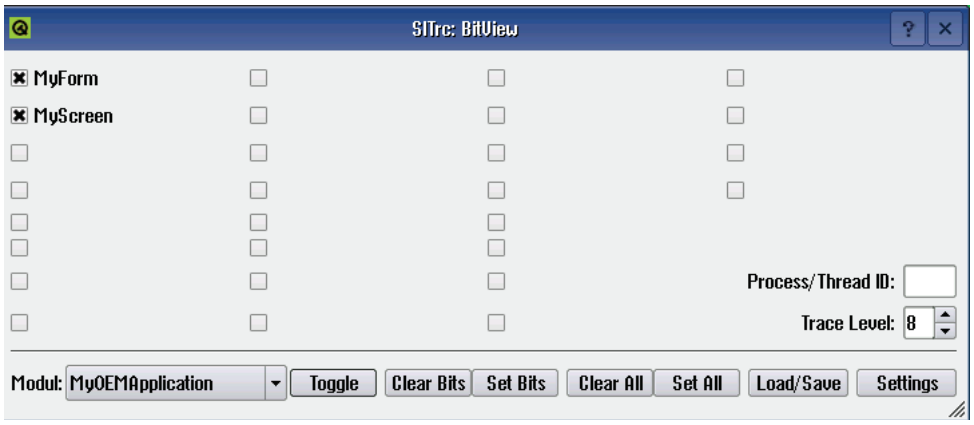


图 4-17:TRACE 模块的设置

选择 TRACE 位后按下“Toggle”切换到 SITrc 模块的位视图。
注意，在该位视图中位“! tracing off”并未选中。在第三列中可以选择一个或多个输出目标。

- TRACE 文件
- TRACE 视图（显示 TRACE 输出的窗口）
- IDE（Visual Studio 中的输出窗口）
- sltrcsvc 的 Stdout

TRACE 文件的保存地点可通过按钮“Settings”设置。

结束 TRACE 后，最好按下按钮“Clear All”，以删除所有 TRACE 位并置位“! tracing off”。

“Interface Trace”和“Preferred Trace”软键均为内部功能，不可使用。

启动 Linux-Embedded 条件下的 HMI

在 Putty 中写入下述命令行启动 HMI：

```
cd /siemens/sinumerik/hmi/autostart
./run_hmi -start slsmsystemmanager -qws -display VNC:0
```

HMI 在 VNC 显示屏 0 上启动，因此最好在相连的薄型客户端上显示。

4.22 OEM 跳转软键

通常情况下 HMI OA 对话框是通过区域切换栏的一个自定义软键选中的。借助 OEM 跳转软键，HMI OA 对话框也可以从 HMI 标准对话框中（比如：“加工”区）选中。

步骤

在每个 HMI 标准对话框中都预留了一个软键供 OEM 用户使用，即 OEM 跳转软键。将来该软键也不会被标准功能占用。

如需在按下预留软键后浏览到到自定义的 HMI OA 对话框，您需要编辑软键对应的模板。模板的保存路径为：

```
[安装目录]\hmisl\siemens\sinumerik\hmi\template\cfg\
```

表 4-231：预留软键/模板一览

HMI 标准对话框	预留软键	模板
加工	ETC0，水平软键 6	slmachine_oem.xml
参数	ETC0，水平软键 7	slparameter_oem.xml
程序	ETC1，水平软键 7	slprogramedit_oem.xml
程序管理器	ETC0，水平软键 7	slpmdialog_oem.xml
诊断	ETC0，水平软键 7	sldiagnose_oem.xml
调试	ETC0，水平软键 7	slsudialog_oem.xml

模板是对应 HMI 标准对话框配置文件的选段。 每份模板中都通过注释指出了允许修改的参数。

以“参数”区为例（允许的修改突出显示）：

```
<DIALOG>
  <TEXTFILE>oemtextfile</TEXTFILE>
  <MENU name="SlPaMenuHU">
    <ETCLEVEL id="0">
      <SOFTKEYGROUP name="SlPaSoftkeyGroup" toggleselectedsoftkey="true" >
        <SOFTKEY position="7" accesslevel="5">
          <PROPERTY name="textID" type="QString">OEM</PROPERTY>
          <PROPERTY name="translationContext"
                    type="QString">OEMContext</PROPERTY>
          <NAVIGATION target="area">
            <AREA name="OEMArea" dialog="OEMDialog" screen="OEMScreen"/>
          </NAVIGATION>
        </SOFTKEY>
      </SOFTKEYGROUP>
    </ETCLEVEL>
  </MENU>
</DIALOG>
```

需要在 HMI OA 对话框中通过按下经过定义的“回调”键回到先前调用它的 HMI 标准对话框（比如：“参数”区）时，NAVIGATION 标签的内容应为：

```
<NAVIGATION target="area" args="-slGfwDynamicArea AreaParameter">
  <AREA name="OEMArea"/>
</NAVIGATION>
```

如果在“systemconfiguration.ini”中只是将某个 HMI OA 对话框作为对话框配置，而不是作为区域配置，现在要跳转到该对话框，则 NAVIGATION 标签的内容应为：

```
<NAVIGATION target="dialog">
  <DIALOG name="SlParameter"/>
</NAVIGATION>
```

这种方式单凭 OEM 跳转软键即可进入自定义的 HMI OA 对话框，而不需要凭借区域切换条中的软键进入。

注

“Recall”配置的 HMI 标准范围（Areas）或 HMI 标准对话框名称参见文件：

```
[Install-Dir]\hmisl\siemens\sinumerik\hmi\cfg\systemconfiguration.ini
```

将模板传送到系统中

将修改后的模板传送到系统的以下目录中：

```
[安装目录]/oem/sinumerik/hmi/appl
```

重启 SINUMERIK Operate 后，对应 HMI 标准对话框中的软键即可投入使用。

注

重启 SINUMERIK Operate 后，您会发现在同一目录下有一份自动生成的 HMI 文件，比如，在使用“slsudialog_oem.xml”时会生成一份“slsudialog.hmi”。如需删除 OEM 跳转软键的定义，必须删除该文件和复制后经过修改的模板。

4.23 FAQ

本章为您解答关于 GUI Framework 和 GUI 组件的常见问题。

对话框类和屏幕类必不可少吗？

程序运行时，对话框类和屏幕类必不可少，但是您无须实现自定义的 HMI 对话框类和屏幕类。如果没有您希望改写的功能，最好使用对话框和屏幕的基本类。

在 XML 配置文件中，您可以保持属性“implementation”为空或指定基本类。

```
<!DOCTYPE HMI DIALOG CONFIGURATION>
<DIALOGUI implementation="slgfw.SlGfwHmiDialog"
         ...>

<SCREEN implementation="slgfw.SlGfwScreen" name="screenMain">
<FORM implementation=...
</SCREEN>
</DIALOGUI>
```

对话框和屏幕的基本类不需要使用其他插件宏命令。此时指定自定义的窗体类即可。

如何得知当前窗体显示在哪个屏幕中？

有两种途径可解决该问题：

1. 利用 GUI Framework 在窗体类 SlGfwDialogForm 中提供的两个虚拟方法：一个是方法 attachedToScreen()，在窗体确实在屏幕中可见时调用。另一个方法是 detachedFromScreen()，在窗体不再可见时调用。方法中屏幕的名称是作为参数传送的。
2. 利用窗体类 SlGfwDialogForm 中的方法 attachedScreen()直接查询屏幕的名称。如果窗体不显示在任何屏幕中，则返回 Nullstring (QString::null)。

何时必须调用 onFunction()的基本实现？

在 onFunction 实现中没有编辑定义的功能时，必须调用它的基本实现。

```
void TestForm::onFunction(const QString& rszFunction,
                          const QString& rszArgs,
                          bool & rbHandled)
{
    if ("func1" == rszFunction)
    {
        //... do something
        rbHandled = true;
    }
    else if ("func2" == rszFunction)
    {
        //... do something
        rbHandled = true;
    }
    else
    {
        SlGfwDialogForm::onFunction(rszFunction, rszArgs, rbHandled);
    }
}
```

为什么在按下软键 OK/Cancel/Accept/Back 后不显示语言文件中的文本，而是显示文本 ID？

为此类软键重新设置文本 ID 后，必须一同重新设置文本上下文，仅设置上级屏幕中所需的文本上下文是不够的。

```
<SOFTKEY_OK position="1">
    <PROPERTY name="textID" type="QString">是</PROPERTY>
    <PROPERTY name="translationContext" type="QString">myContext</PROPERTY>
</SOFTKEY_OK>
```

为什么在系统启动后不显示对话框，屏幕黑屏？

针对这种情况，建议您检查下面几点：

1. 是否有屏幕布局文件？

```
<!DOCTYPE HMI_DIALOG_CONFIGURATION>
<DIALOGUI implementation="slgfw.SlGfwHmiDialog"

   screenlayout="slstandardscreenlayout.SlStandardScreenLayout">
...
</DIALOGUI>
```

2. 屏幕布局文件中是否有窗体面板？

```
<!DOCTYPE HMI_DIALOG_CONFIGURATION>
<DIALOGUI implementation="slgfw.SlGfwHmiDialog" …>
    <SCREEN name="screenMain">
        <FORM implementation=... formpanel="FullForm"/>

           </SCREEN>
</DIALOGUI>
```

3. 窗体的实现是否正确？

```
<!DOCTYPE HMI_DIALOG_CONFIGURATION>
<DIALOGUI implementation="slgfw.SlGfwHmiDialog" …>
    <SCREEN name="screenMain">
        <FORM implementation="sltest.TestForm" .../>
                ...
    </SCREEN>
</DIALOGUI>
```

4. 是否将窗体类通知了库中的插件机制？

```
#include "testform.h"

SL GFW BEGIN PLUGIN EXPORT()
    SL_GFW_DIALOGFORM_PLUGIN_EXPORT(testForm)
    ...
SL_GFW_END_PLUGIN_EXPORT()
```

在切换操作区域时，屏幕和窗体的事件按什么顺序出现？

查看“GUIFrameWork\SlExGuiEvents”下示例目录中的“Event-Logger”示例。 该示例指出了屏幕和窗体所有事件的时间顺序。

如何借助 GUI Framework 设计出一个后台应用程序？

查看“GUIFramework\SlExGuiBackgroundApplication”下的示例。重要步骤：

1) 在“Systemconfiguration.ini”中添加自定义对话框。但此处不要为该对话框定义任何操作区域(Area)。
2) 设置参数 preload:=true 加载对话框，使对话框随系统一同启动。
3) 如果该对话框也包含了自定义的屏幕或窗体，也要在对话框配置文件中通过设置参数 preload:=true 加载这些屏幕或窗体。

```
<!DOCTYPE HMI_DIALOG_CONFIGURATION>
<DIALOGUI defaultscreen="realtimescreen"
          screenlayout="slstandardscreenlayout.SlStandardScreenLayout"
          panellayout="StandardLayout">
  <SCREEN implementation="sloem1.realtimescreen" name="realtimescreen"
          preload="true">
    <FORM implementation="sloem1.realtimeform" name="realtimeform"
          formpanel="FullForm" preload="true"/>
      ...
  </SCREEN>
</DIALOGUI>
```

如何禁用或隐藏 SIGrGrid 小部件的标题？

可调用“setLeftMargin(0)”或“setTopMargin(0)”隐藏垂直标题列（垂直标题）或水平标题行（水平标题）。

5

5 与 NC/PLC 的通讯

本章主要内容

本章介绍 CAP 服务的接口。它应用在应用程序与 NC/PLC 之间的通讯中。

注

通讯没有时间保障。因此 SINUMERIK Operate 编程包不能解决实时任务。

5.1 引言

5.1.1 类模型

概述

CAP 服务的类模型主要由以下的类组成：

- SlQCap
- SlQCapHandle
- SlQCapNamespace

SlQCap 类

访问 NC 和 PLC 上的数据是通过 SlQCap 对象实现的。特别是该对象可以读/写变量，在值变化时发出通知。另外对象还可以传送文件，启动 PI 服务。

几乎所有的的调用既可以采取同步方式，也可以采取异步方式。其中，同步调用会阻塞调用所在的线程，直到任务完成。而在异步调用中，SlQCap 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

只存在默认构造函数。无法复制 SlQCap 对象或对其赋值。

SlQCapHandle 类

在所有异步调用或触发 Qt 信号的调用中，都会返回 SlQCapHandle 类的一个对象。该类的各个对象可以标识出对应的任务。

删除 SlQCapHandle 的对象会取消对应的任务。

可以复制并赋值该类的对象以及检查对象是否相等。

SlQCapNamespace 类

SlQCapNamespace 类的对象可以向 CAP 服务声明机床专用的数据访问（诸如机床数据、全局用户数据 GUD 等）。这些声明在 SlQCapNamespace 对象的整个寿命内都可用。

注

SlQCap、SlQCapHandle 和 SlQCapNamespace 的对象只允许在 Qt 主线程中使用。

但函数 ***read***、***write***、***mulitRead*** 和 ***multiWrite*** 除外。
这些函数也允许在辅助线程（worker thread）中使用。前提是在堆（heap）中建立了必要的 SlQCap 对象。

5.1.2 术语解释

同步调用

同步调用在任务执行完后才会返回，即当前线程会被一直阻塞。这会干扰事件处理，例如在同步调用时阻止输入和显示。因此比较耗时的调用应进行异步调用。

异步调用

一旦任务成功发送给 CAP 服务，
异步调用和 Hotlink 就会返回。这尤其意味着，所提供的故障代码不代表任务是否成功完成，而只是指出任务是否成功发送。例如当未正确提供调用参数时，则会出错。真正的任务状态在回调 CAP 服务时提供（信号与槽机制）。

注

向 CAP 服务发送多个异步调用时，无法确保任务按照发送的先后顺序依次执行。

单个调用

在单个调用时，只对一个变量进行读/写访问。有以下可用的单个调用：

表 5-1：单个调用

SIQCap 调用	描述
read	同步读取一个变量
write	同步写入一个变量
readAsync	异步读取一个变量
writeAsync	异步写入一个变量

多变量调用

在多变量调用时，可访问多个变量。任务的变量此时可进行完全不同的数据寻址。
有以下可用的多变量调用：

表 5-2：多变量调用

SIQCap 调用	描述
multiRead	同步读取多个变量
multiWrite	同步写入多个变量
multiReadAsync	异步读取多个变量
multiWriteAsync	异步写入多个变量

Hotlink

Hotlink 指在一个或多个变量数值变化时发出通知。通知采用“信号与槽机制”。
首次通知提供的是建立 Hotlink 时的值。

只要 CAP 服务一识别到数值变化，就会发送通知。此时会丢失中间值，而不是终值。

有以下可用的调用：

表 5-3：Hotlink 调用

SIQCap 调用	描述
advise	一个变量变化时发出通知
multiAdvise	多个变量变化时发出通知
unadvise	删除通知
suspend	暂停通知
resume	恢复通知

数组访问

对于一些变量，可在其变量路径下定义多个连续的变量。这称之为
数组访问。

数组访问加快了数据访问并可因此提高整个系统的速度，因为通讯时间被大大缩短了。

例如对三个连续的 R 参数（R7、R8、R9）的数组访问可为：

```
"/channel/parameter/r[u1,7,#3]"
```

PI 服务

PI 服务是一个在 NC 或 PLC 中执行的程序实例服务（program instance，简称 PI）。有以下可用的调用：

表 5-4：PI 调用

SIQCap 调用	描述
piStart	同步启动 PI 命令
piStartAsync	异步启动 PI 命令

允许的PI服务参见PI帮助手册。该帮助手册
也是 SINUMERIK Operate 编程包文档的一部分。

此处需要注意的是，“/NC”和真正的 PI 服务组合成了命令，所有变量在第二个参数中传递。

PI 帮助手册中的示例：
PI_START(/NC, 001, TESTWORD, _N_LOGIN_)

命令为“/NC/_N_LOGIN_”
参数为“001”和“TESTWORD”

跨域传送（domain transfer）

跨域传送指将文件（如零件程序）传送到 NC 中（即下载）或将域从 NC 传送到 HMI 的文件中（即上传）。有以下可用的调用：

表 5-5：跨域传送

SIQCap 调用	描述
download downloadNc	同步下载文件
downloadAsync downloadNcAsync	异步下载文件
upload uploadNc	同步上传文件
uploadAsync uploadNcAsync	异步上传文件

管道式跨域传送（Piped domain transfer）

管道式跨域传送指将数据流传送到 NC 中的一份文件中（即下载）或将数据流从 NC 传送到 HMI 的文件中（即上传）。有以下可用的调用：

表 5-6：管道式跨域传送（Piped domain transfer）

SIQCap 调用	描述
download downloadNc	同步下载管道
downloadAsync downloadNcAsync	异步下载管道
upload uploadNc	同步上传管道
uploadAsync uploadNcAsync	异步上传管道
writePipe	将数据同步写入管道中
writePipeAsync	将数据异步写入管道中

这些调用和“跨域传送”的名称一致，只不过填入了其他调用参数（装载函数）。

5.2 互动测试接口

一览

编程包中提供了一个互动式测试程序“slcaptest.exe”，以便您访问数据，初步了解 CAP 服务接口的工作方式。其可监控应用程序的编程，例如变量，或者检查其存在性。

应用

测试程序通过以下开始菜单项来启动：

```
开始 →程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create
MyHMI -3GL →工具 →slCapTest (-colocated)
```

参数-colocated 的作用是同时加载和启动 CAP-Service，否则无法进行通讯。此外还需启动：

```
开始 →程序 → Siemens Automation → Sinumerik → SINUMERIK Integrate Create
MyHMI -3GL →工具 →cp_840Di
```

注

必须适当地修改文件“\hmisl\oem\sinumerik\hmi\cfg\mmc.ini”。

在首次启动后，初始画面为一个含有多个行的页面，可分别在其中执行对 CAP-Service 的访问。该页面名为“DefaultTab”并属于“DefaultGroup”组。通过按钮“Add Tab...”可继续添加页面和组。这样可将不同的测试场景更好地加以区分。

退出测试程序时，会自动保存当前设置的参数。

按钮

画面底边上的按钮的作用如下：

1. “AddTab ...”
   在一个组中创建新的页面（标签）。如果所输入的组还不存在，则会生成一个新组。名称只能由字母、数字和“_”组成。

2. “DelTab”
   删除当前显示的页面及其输入栏。如果所删除的页面是该组的最后一页，则会删除整个组。

3. “Clear”
   删除所显示页面的所有输入栏并将按钮恢复为初始状态。

4. “DoitAll”
   只要变量路径不为空，会一次按下所显示页面上的所有按钮。此时按钮被按下的顺序为从上至下。

画面底部末端的列表框(listbox）可进行组的选择。

指令

通过多次按下每行的首个按钮可以选择一种操作。有以下几种选择：

1. “Hotlink”
   在每行左侧的输入栏中输入一个变量路径。然后通过按下按钮“Start”建立 Hotlink，按下按钮“Stop 可再次中断该链接。如果 Hotlink 生效，会在右侧的输入栏中显示当前值。

2. “Read”
   在每行左侧的输入栏中输入一个变量路径。之后按下按钮“Doit”可启动对变量的异步读取调用。所读取的值显示在右侧的输入栏中。

3. “Write”
   在每行左侧的输入栏中输入一个变量路径，在右侧的输入栏中输入要写入的值。之后按下按钮“Doit”可启动异步写入调用。

如果在执行调用时出错，会在两个输入栏的右侧显示十六进制的故障代码。“Read”和“Hotlink”操作此时会显示值“#”。

注

如将鼠标光标放在十六进制故障代码的显示区域上，会出现显示在提示框(Tooltip)中的相应故障文本。如将鼠标光标放在读取的值上，则会显示附加信息。

示例

以下为测试程序“slcaptest.exe”的几个应用示例：

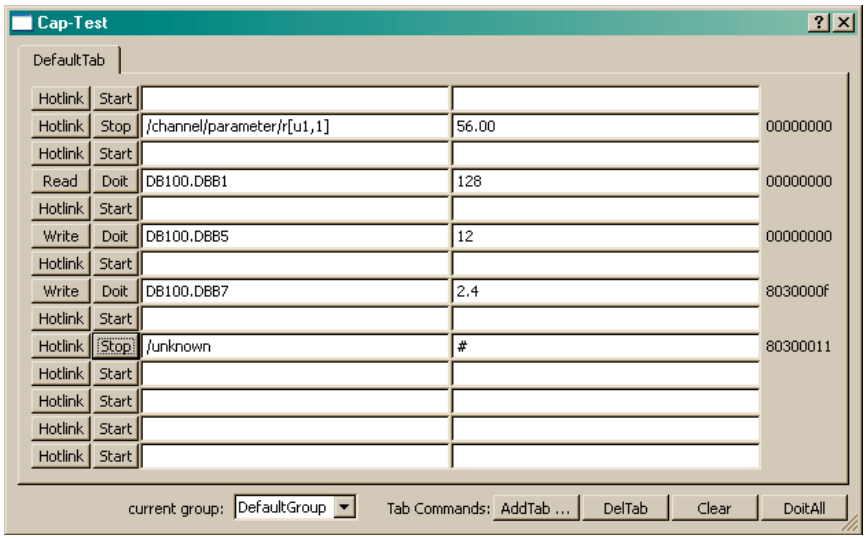


图 5-1:“slcaptest.exe”的初始画面

在图 5-1：“slcaptest.exe”初始画面中可找到各个点的说明：

1. 启动了一个 Hotlink，用来监控第一个通道的 R 参数 R[1]。当前值为 56。如果值发生变化，显示也会变化。
2. 一次读取了数据块 100 的第二个字节。读取的值为 128。
3. 成功将值 12 写入数据块 100 的第六个字节中。
4. 尝试向一个字节中写入浮点数。写入失败，返回值为 8030000f。
5. 尝试在未知变量上建立 Hotlink。操作失败，返回值为 803000011。

5.3 分步示例

概述

以下章节将分步介绍 CAP 服务的不同应用区域。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例。

以下为这些示例和所有其他示例一览。

表 5-7：示例一览

应用	方式	示例	章节
读/写一个变量	同步	slexcapsyncreadwrite	5.3.2
	异步	slexcapasyncreadwrite	5.3.3
读/写多个变量	同步	slexcapmultisyncreadwrite	5.3.4
	异步	slexcapmultiasyncreadwrite	-
与一个变量的 Hotlink		slexcapdatachange	5.3.5
与多个变量的 Hotlink		slexcapmultidatachange	-
数组访问	同步	slexcapsyncarrayaccess	5.3.6
启动 PI 命令	同步	slexcapsyncpi	5.3.7
	异步	slexcapasyncpi	5.3.8
机床数据/GUD 的声明	同步	slexcapsyncmap	5.3.9
	异步	slexcapasyncmap	-
传送零件程序	同步	slexcapsyncdomaintransfer	5.3.10
	异步	slexcapasyncdomaintransfer	5.3.11
通过 pipe 将数据传送到零件程序中	同步	slexcapsyncpipeddomaintransfer	-
	异步	slexcapasyncpipeddomaintransfer	-

此处只举例说明或描述与 CAP 服务相关的内容。源代码的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

开发环境

如需直接在 PC 上测试示例程序，需要将 PC 和 Sinumerik 840D sl 系统连接在一起。

在生成示例程序后，所需的文件位于“debug/output ”或“release/output”中。子目录 (appl, cfg, lng, hlp)使目标目录中文件的保存情况一目了然。

- \appl\[Beispielname].hmi
- \appl\[Beispielname].dll
- \cfg\systemconfiguration.ini

您可以激活事先准备好的“Post-Build Event”，以便直接将生成的文件复制到您 PC 的 OEM 目录 (\hmisl\oem\sinumerik\hmi\)下。它适用于所有可执行的示例程序。激活“Post-Build Event”的步骤如下：

1. 菜单“Project”
2. 菜单项“[Example name] Properties...”
3. 浏览到“Configurations Properties/Build Events/Post-Build Event”
4. 将“Excluded From Build”设为“No”。

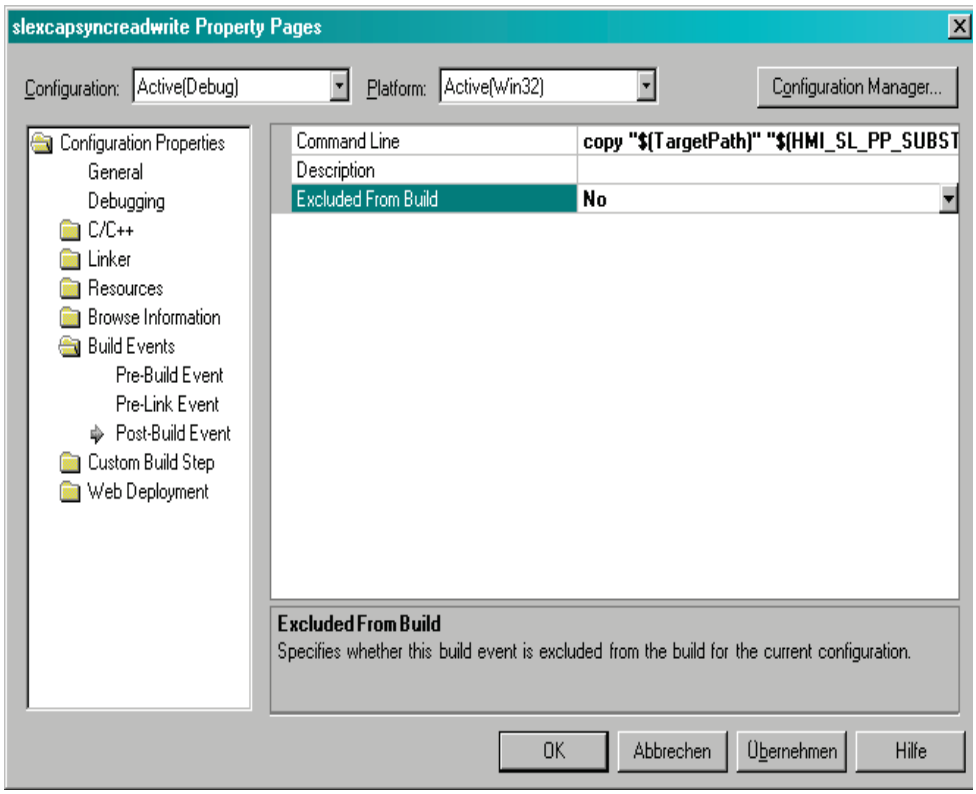


图 5-2:激活“Post-Build Event”

接着建议您根据您的实际需要来修改“systemconfiguration.ini”文件（Dialog & Area Index）。然后将该文件复制到 OEM-CFG 目录(\hmisl\oem\sinumerik\hmi\cfg)中。该文件不会自动复制，以避免改写可能存在的旧配置文件。

注

为确保 SINUMERIK Operat 编程包中示例程序的格式正确，建议您将 Visual Studio 中的“Tab size”和“Indent size”分别设为 4，设置菜单为：*Tools → Options → Text Editor → C++ → Tabs*。

5.3.1 准备

概述

满足以下前提后，才能从自定义的项目或类中调用 CAP 服务。这些条件同样也针对下文中的所有“分步示例”。

检查库

在项目设置中检查库 `slcap.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `slcap.lib`

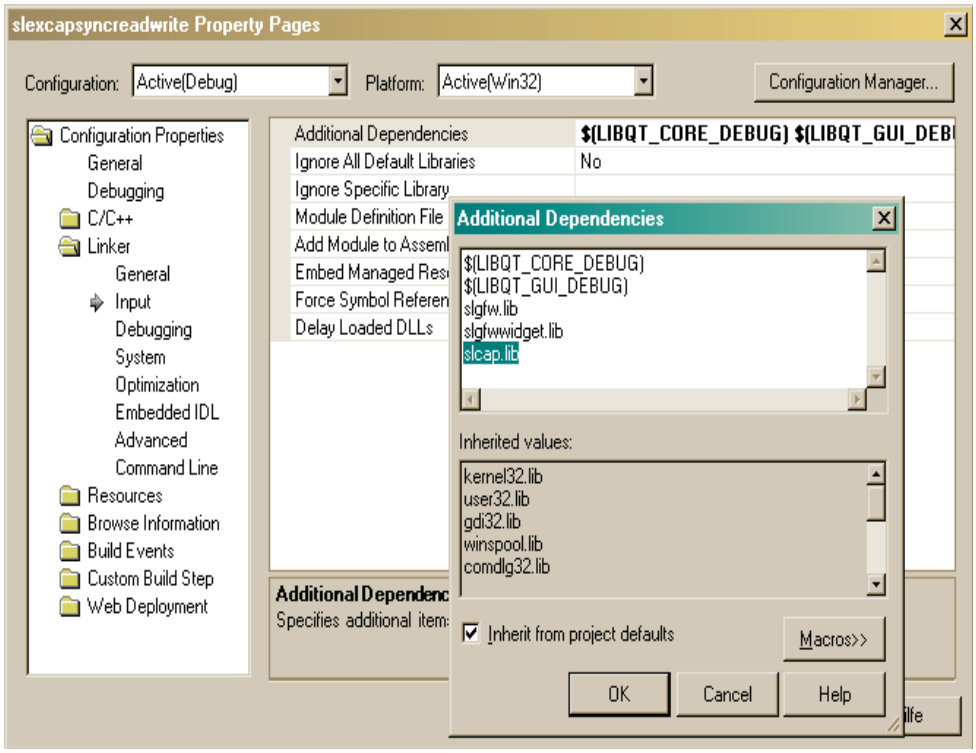


图 5-3:库“slcap.lib”

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lslcap
```

头文件

要使用 CAP 服务的类需要加入头文件“slqcap.h”，配置条目为：

```
#include "slqcap.h"
```

创建 SlQCap 对象

访问 CAP 服务的接口需要使用 SlQCap 类的对象。这些对象可作为私有的成员变量创建。
在创建时要注意，SlQCap 类的一个对象始终只能执行一个任务，也就是说，需要同时处理多个异步访问时，需要使用多个 SlQCap 对象。因此在设置 Hotlink 后对应的 SlQCap 对象被占用，不可用于读写访问。

示例：一个用于同步访问的 SlQCap 对象

```
SlQCap m_capServer;
```

示例：两个 SlQCap 对象，一个用于同步读写，一个用于 Hotlink

```
SlQCap m_capServerReadWrite;
SlQCap m_capServerHotlink;
```

注

SlQCap 类的对象只有少数的实例数据，在创建和删除时费时较短。因此 SlQCap 对象也可以在栈上创建。

5.3.2 同步读/写一个变量

概述

下面的示例分步介绍了如何同步读一个变量并接着同步写一个变量。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExCapSyncReadWrite”。

在示例中输入一个 item，然后通过软键“read data”读 item 或“write data”写 item。状态栏显示调用成功。

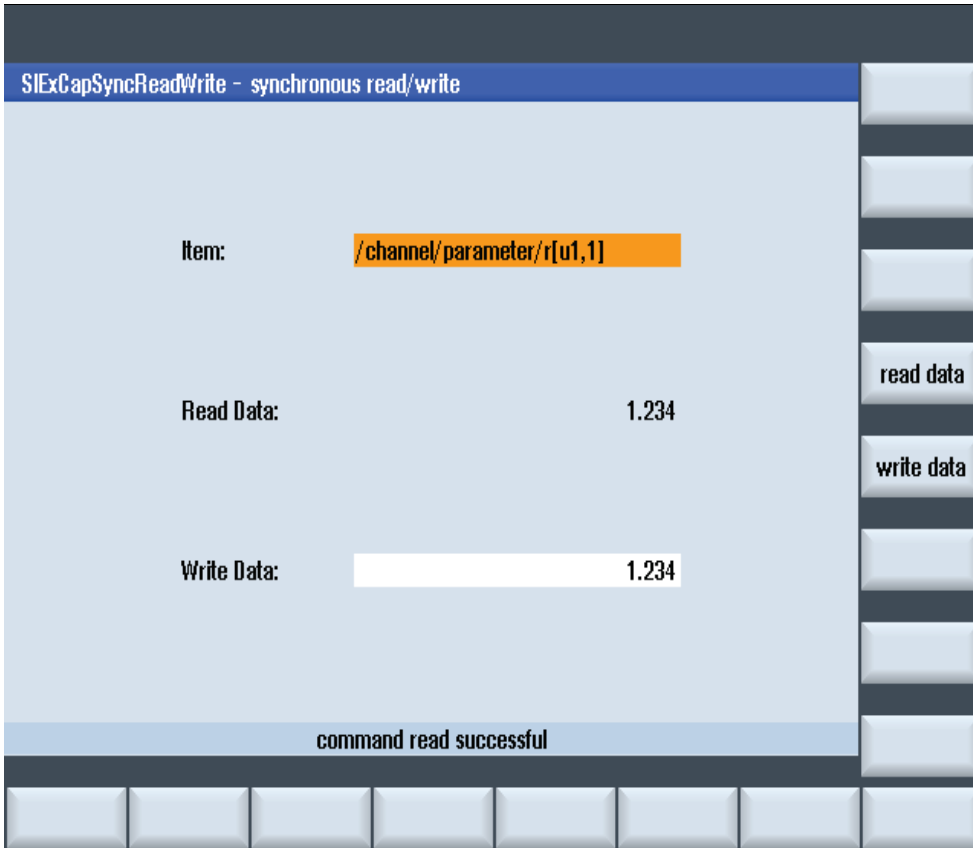


图 5-4:示例 SIExCapSyncReadWrite

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

创建一个 QVariant 变量，读出的结果值便保存在该变量中。

```
    QVariant vData;
```

第 2 步

读出第一通道中的 R 参数 R[1]，向 QVariant vData 中写入读出的值。该任务的执行状态写入变量 eError 中。

```
QString szItem = "/channel/parameter/r[u1,1]";
SlCapErrorEnum eError = m_capServer.read(szItem, vData);
```

第 3 步

查看读任务的执行状态。

```
if( SL_CAP_OK  != eError )
{
      // 插入故障处理
}
else
{
      // vData 包含了读出的值，用于后续处理
}
```

第 4 步

向 QVariant 赋值，执行写变量任务。

```
QVariant vData(1.2345);
或者
QVariant vData = 1.2345;
```

第 5 步

下一条调用将 vData 的内容(1.2345)写入第二通道的 R 参数 R[5]中。该任务的执行状态写入变量 eError 中。

```
QString szItem = "/channel/parameter/r[u2,5]";
SlCapErrorEnum eError = m_capServer.write(szItem, vData);
```

第 6 步

查看写任务的执行状态。

```
if( SL CAP OK  != eError )
{
          // 插入故障处理
}
```

5.3.3 异步读/写一个变量

概述

下面的示例分步介绍了如何异步读一个变量并接着异步写一个变量。 SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目 “SlExCapAsyncReadWrite”。该示例程序的结构与 5.3.2 章描述的同步读/写一个变量的示例相同。

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

首先，作为私有成员变量创建 SlQCapHandle 类的两个对象，因为一个 SlQCap 对象始终只能执行一个任务。通过创建多个对象可以同时执行多个异步任务。

除了这两个对象以外，还需要创建 SlQCapHandle 类的另外两个对象。其中第一个标记读任务，第二个标记写任务。

```
SlQCap m capServerRead;
SlQCap m capServerWrite;

SlQCapHandle m_capHandleRead;
SlQCapHandle m_capHandleWrite;
```

第 2 步

任务的执行结果通过信号通知 CAP 服务。这些信号由以下函数（槽）接收。这两个函数的定义在第 5 步(`readDataSlot`)和第 7 步(`writeCompleteSlot`)中变得清晰明确。

```
private slots:
         void readDataSlot(SlCapErrorEnum, const QVariant&,
                    const SlCapSupplementInfoType&);

         void writeCompleteSlot(SlCapErrorEnum, const QDateTime&);
```

第 3 步

将成员函数（槽）与对应的 CAP 服务发出的信号关联在一起。

```
QObject::connect(&m capServerRead,
                        SIGNAL(readData(SlCapErrorEnum, const QVariant&,
                                         const SlCapSupplementInfoType&)),
                        this,
                        SLOT(readDataSlot(SlCapErrorEnum, const QVariant&,
                                        const SlCapSupplementInfoType&)));

QObject::connect(&m capServerWrite,
            SIGNAL(writeComplete(SlCapErrorEnum, const QDateTime&)),
            this,
            SLOT(writeCompleteSlot(SlCapErrorEnum, const QDateTime&)));
```

第 4 步

向 CAP 服务传送读任务：读出第一通道中的 R 参数 R[1]。

```
QString szItem = "/channel/parameter/r[u1,1]";
SlCapErrorEnum eError = m_capServerRead.readAsync(szItem, &m_capHandleRead);
```

第 5 步

异步调用立即返回。任务的执行结果位于槽 readDataSlot 中。

```
void SlExCapAsyncReadWriteForm::readDataSlot(
                                        SlCapErrorEnum eError,
                                        const QVariant& rvData,
                                        const SlCapSupplementInfoType&)
{
      if( SL_CAP_OK  != eError )
      {
               // 插入故障处理
      }
      else
      {
                                           // rvData 包含了读出的值，用于后续处理
      }
}
```

第 6 步

向 CAP 服务传送写任务：将 vData 的内容(1.76)写入第三通道的 R 参数 R[7]中。

```
QVariant vData(1.76);
QString szItem = "/channel/parameter/r[u3,7]";
SlCapErrorEnum eError = m_capServerWrite.writeAsync(szItem, vData,
                                                    &m_capHandleWrite);
```

第 7 步

异步调用立即返回。但写任务在调用槽 writeCompleteSlot 后才完成。

```
void SlExCapAsyncReadWriteForm::writeCompleteSlot(SlCapErrorEnum eError,
                                                    const QDateTime&)
{
      if( SL_CAP_OK  != eError )
      {
            // 插入故障处理
      }
}
```

5.3.4 同步读/写多个变量

概述

下面的示例分步介绍了如何同步读多个变量并接着同步写多个变量。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExCapMultiSyncReadWrite”。

在示例中输入两个相互独立的 item，然后通过软键“read data”或“write data”对其进行读或写。状态栏显示调用成功。

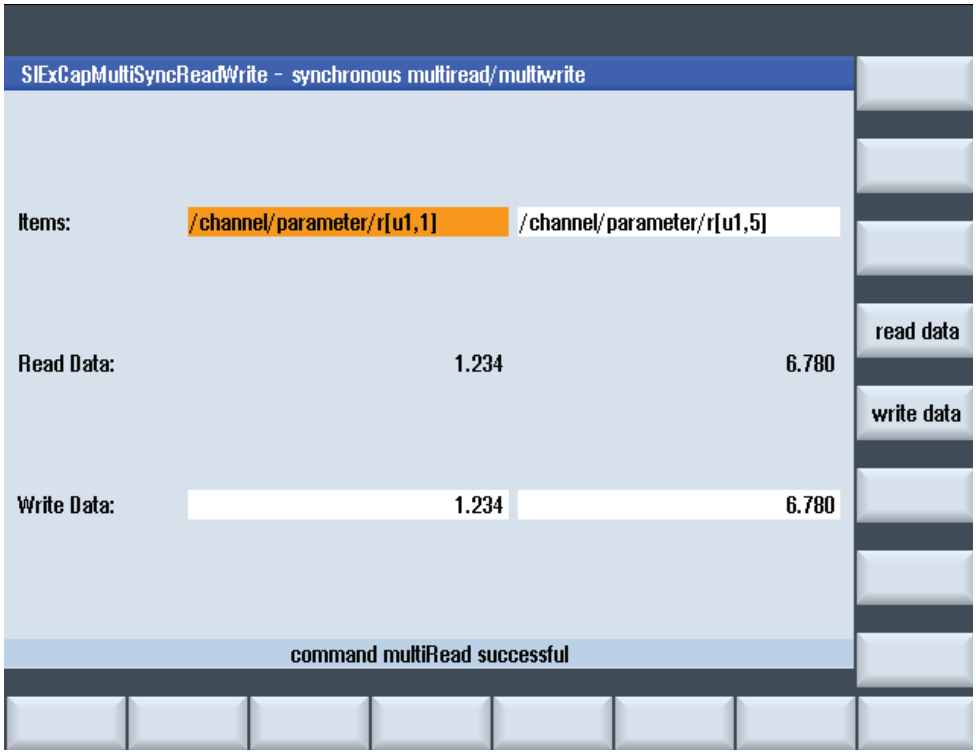


图 5-5:示例 SIExCapMultiSyncReadWrite

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

创建数据类型为 SICapReadSpecType 的一个 QVector。待读出的数据中要插入到该矢量中。在本例中待读出的数据是第一通道的 R 参数 R[1]和第二通道的 R 参数 R[5]。

```
QString szItem1 = "/channel/parameter/r[u1,1]";
QString szItem2 = "/channel/parameter/r[u2,5]";

QVector<SlCapReadSpecType> vecReadSpec;
vecReadSpec.append( SlCapReadSpecType(szItem1) );
vecReadSpec.append( SlCapReadSpecType(szItem2) );
```

第 2 步

创建数据类型为 SICapReadResultType 的一个 QVector。读出的结果值要保存到该矢量中。

```
QVector<SlCapReadResultType> vecReadResult;
```

第 3 步

读出 QVector`vecReadSpec` 中确定的变量，向 QVector `vecReadResult` 写入读出的值。整个读任务的执行状态写入到变量 `eError` 中，单个变量读任务的执行状态写入到 QVector`vecReadResult` 中。

```
SlCapErrorEnum eError = m_capServer.multiRead(vecReadSpec, vecReadResult);
```

第 4 步

查看读任务的执行状态。所有单个变量的读任务成功执行后，返回 SL_CAP_OK。出错时可以查看单个变量的读错误。

```
if( SL_CAP_OK  != eError )
{
      // 插入故障处理
      SlCapErrorEnum eKanal1_R1 = vecReadResult[0].m_eError;
      SlCapErrorEnum eKanal2_R5 = vecReadResult[1].m_eError;
}
else
{
      // vecReadResult 包含了读出的数值，用于后续处理
}
```

第 5 步

可以按照索引号来查看读出的单个值。结果和 QVariant 调用单个变量时一样。

```
QVariant vKanal1_R1 = vecReadResult[0].m_vValue;
QVariant vKanal2_R5 = vecReadResult[1].m_vValue;
```

第 6 步

创建数据类型为 SlCapWriteSpecType 的一个 QVector。待写入的数据中要插入到该矢量中。在本例中，值 1.23 要写入到第一通道的 R 参数 R[1]中，值 77 要写入到第二通道的 R 参数 R[5]中。

```
QVariant vKanal1_R1 = 1.23;
QVariant vKanal2_R5 = 77;
QString szItem1 = "/channel/parameter/r[u1,1]";
QString szItem2 = "/channel/parameter/r[u2,5]";

QVector<SlCapWriteSpecType> vecWriteSpec;
vecWriteSpec.append( SlCapWriteSpecType(szItem1, vKanal1_R1) );
vecWriteSpec.append( SlCapWriteSpecType(szItem2, vKanal2_R5) );
```

第 7 步

创建数据类型为 SlCapWriteResultType 的一个 QVector。

```
QVector<SlCapWriteResultType> vecWriteResult;
```

第 8 步

写入 QVectorvecWriteSpec 确定的变量和数值。整个写任务的执行状态写入到变量 eError 中，单个变量写任务的执行状态写入到 QVectorvecWriteResult 中。

```
SlCapErrorEnum eError = m_capServer.multiWrite(vecWriteSpec,
                                                &vecWriteResult);
```

第 9 步

查看写任务的执行状态。此处也可以查看单个写任务的错误。

```
if( SL CAP OK  != eError )
{
      SlCapErrorEnum eKanal1 R1 = vecWriteResult[0].m eError;
      SlCapErrorEnum eKanal2 R5 = vecWriteResult[1].m eError;
      ..
}
```

5.3.5 一个变量的 Hotlink

概述

下面的示例将分步介绍如何建立与一个变量的 Hotlink。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExCapDataChange”。

在示例中输入一个 item，然后通过软键“read data”读 item 或“write data”写 item。 通过“start refresh”可以创建一个 Hotlink，通过“stop refresh”可以再次结束该 Hotlink。状态栏显示调用成功。

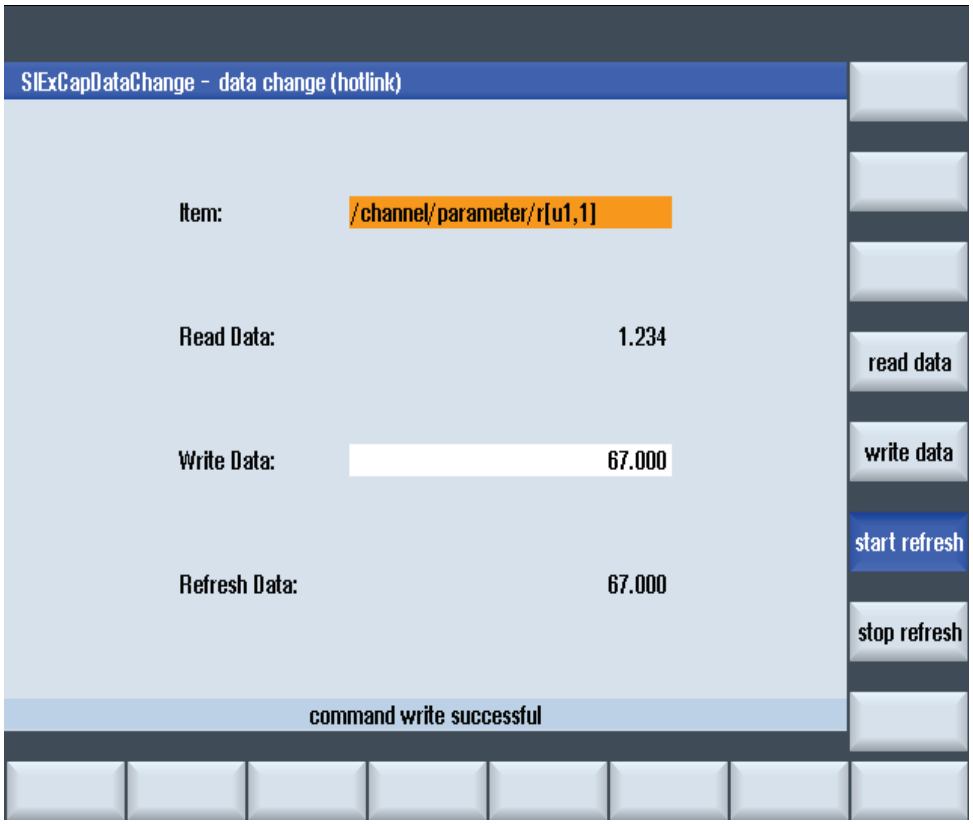


图 5-6:示例 SlExCapDataChange

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

作为私有成员变量创建 SlQCapHandle 类的两个对象。一个 SlQCap 对象用于同步读/写，另一个用于 Hotlink。
除了这两个对象以外，还需要创建 SlQCapHandle 类的另外一个对象。该对象用于标识 Hotlink。

```
SlQCap m capServerReadWrite
SlQCap m capServerHotlink;

SlQCapHandle m_capHandleHotlink;
```

第 2 步

任务的执行结果通过信号通知 CAP 服务。该信号由以下函数（槽）接收。函数的定义在第 5 步(adviseDataSlot)中变得清晰明确。

```
private slots:
      void adviseDataSlot(SlCapErrorEnum, const QVariant&,
                                          const SlCapSupplementInfoType&);
```

第 3 步

现在将成员函数（槽）与 CAP 服务发出的信号 `adviseData`
关联在一起。

```
QObject::connect(&m_capServerHotlink,
                  SIGNAL(adviseData(SlCapErrorEnum, const QVariant&,
                                          const SlCapSupplementInfoType&)),
                  this,
                  SLOT(adviseDataSlot(SlCapErrorEnum, const QVariant&,
                                         const SlCapSupplementInfoType&)));
```

第 4 步

向 CAP 服务传送通知任务：在第一通道的 R 参数 R[1]变化时通知客户。

```
QString szItem = "/channel/parameter/r[u1,1]";
SlCapErrorEnum eError = m_capServerHotlink.advise(szItem,
m_capHandleHotlink);
```

第 5 步

调用立即返回。变量值或值的变化在槽 `adviseDataSlot` 中查看。首次通知提供的是建立时的值。

```
void SlExCapDataChangeForm::adviseDataSlot(
                                                    SlCapErrorEnum eError,
                                                  const QVariant& rvData,
                                          const SlCapSupplementInfoType&)
{
         if( SL_CAP_OK  != eError )
         {
               // 插入故障处理
         }
         else
         {
               // rvData 包含了读出的值，用于后续处理
         }
}
```

第 6 步

不再需要 Hotlink 时要将它删除，以便留出更多的通讯资源。

```
SlCapErrorEnum eError = m_capServerHotlink.unadvise(m_capHandle);
```

5.3.6 数组访问

概述

下面的示例分步介绍了如何实现对一个变量的“数组访问”。SINUMERIK Operate 编程包中有一个展示如何完成此任务的示例程序：项目“SIExCapSyncArrayAccess”。

在示例中输入一个 array item，然后通过软键“read data”读 item 或“write data”写 item。通过“start refresh”可以创建一个 Hotlink，通过“stop refresh”可以次结束该 Hotlink。读出的单个数组元素用符号“|”隔开，同理，在写数组元素时也要用该符号隔开。状态栏显示调用成功。

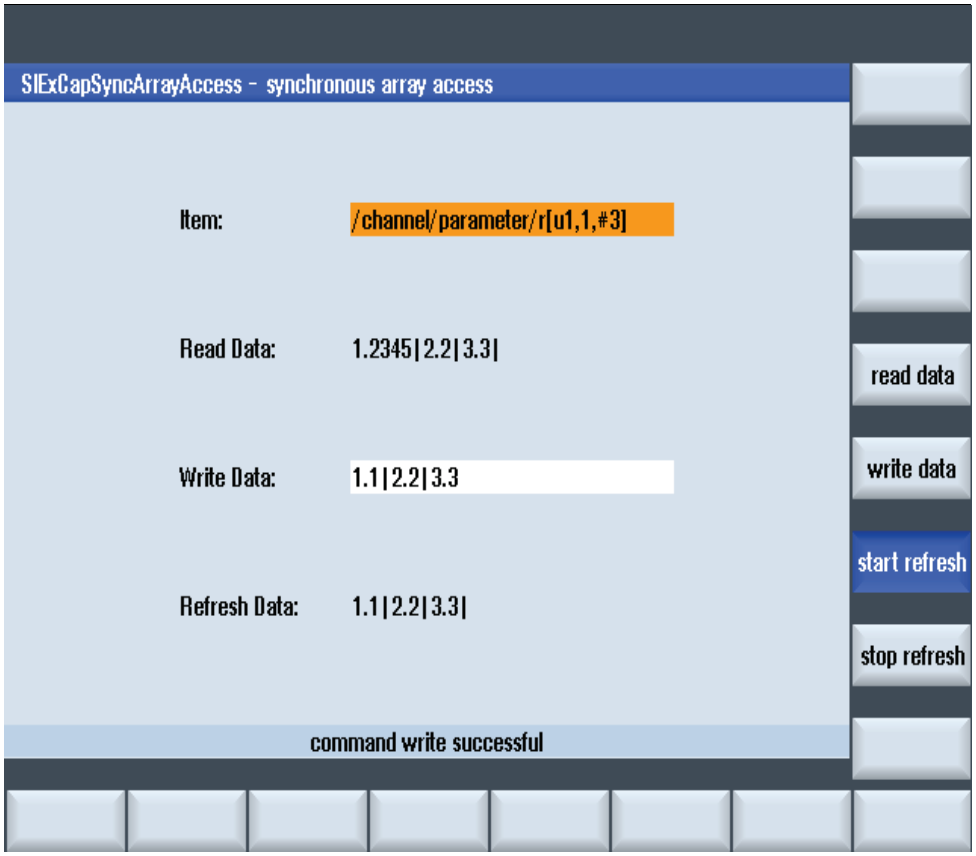


图 5-7:示例 SIExCapSyncArrayAccess

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

创建一个 QVariant 变量，读出的结果值要保存在该变量中。

```
QVariant vData;
```

第 2 步

读出第一通道中的 R 参数 R[1]、R[2]和 R[3]，向 QVariant vData 中写入读出的值。该任务的执行状态写入变量 eError 中。

```
QString szItem = "/channel/parameter/r[u1,1,#3]";
SlCapErrorEnum eError = m_capServerReadWrite.read(szItem, vData);
```

第 3 步

查看读任务的执行状态。

```
if( SL CAP OK  != eError )
{
            // 插入故障处理
}
else
{
            // vData 包含了 QList<QVariant>，用于后续处理
}
```

第 4 步

结果变量 vData 包含了数据类型 QList<QVariant>。通过迭代整个列表可以获得单个数值。在下面的例子中，单个数据是通过 pipe 隔开，封装在一个字符串中。

```
QString szValue     = QString::null;
QVariantList vValue = vData.toList();
for (QVariantList::iterator it = vValue.begin(); it != vValue.end(); it++ )
{
            szValue += (*it).toString();
            szValue += "|";
}
```

...或者直接访问单个数值，比如：

```
double dR1 = vValue[0].toDouble();
double dR2 = vValue[1].toDouble();
double dR3 = vValue[2].toDouble();
```

注

从字节中读取数组（比如：DB123.DBB0:[3）时，结果变量 vData 返回数据类型 “QByteArray”。此时执行以下操作：

```
QByteArray byteArray = vValue.toByteArray();
for ( int nIndex = 0; nIndex < byteArray.count(); nIndex++ )
{
     unsigned char nValue = byteArray[nIndex];
     szValue += QString("%1").arg(nValue);
     szValue += "|";
}
```

第 5 步

向 QList<QVariant>赋值，执行写变量任务。此时也可以接受 QStringList 数组容器（array container）。

```
QString szData = "7|1.234|5"
QStringList szList = szData.split("|");
QVariant vData(szList);
```

第 6 步

将 vData 的内容写入第三通道的 R 参数 R[1]、R[2]和 R[3]中。该任务的执行状态写入变量 eError 中。

```
QString szItem = "/channel/parameter/r[u1,1,#3]";;
SlCapErrorEnum eError = m_capServerReadWrite.write(szItem, vData);
```

第 7 步

查看写任务的执行状态。

```
if( SL CAP OK  != eError )
{
      // 插入故障处理
}
```

5.3.7 同步启动 PI 命令

概述

下面的示例分步介绍了如何同步启动 PI 命令。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExCapSyncPi”。

在本例中，PI 命令是通过软键“run command”启动的。多余的变量用空字符串传递。状态栏显示调用成功。

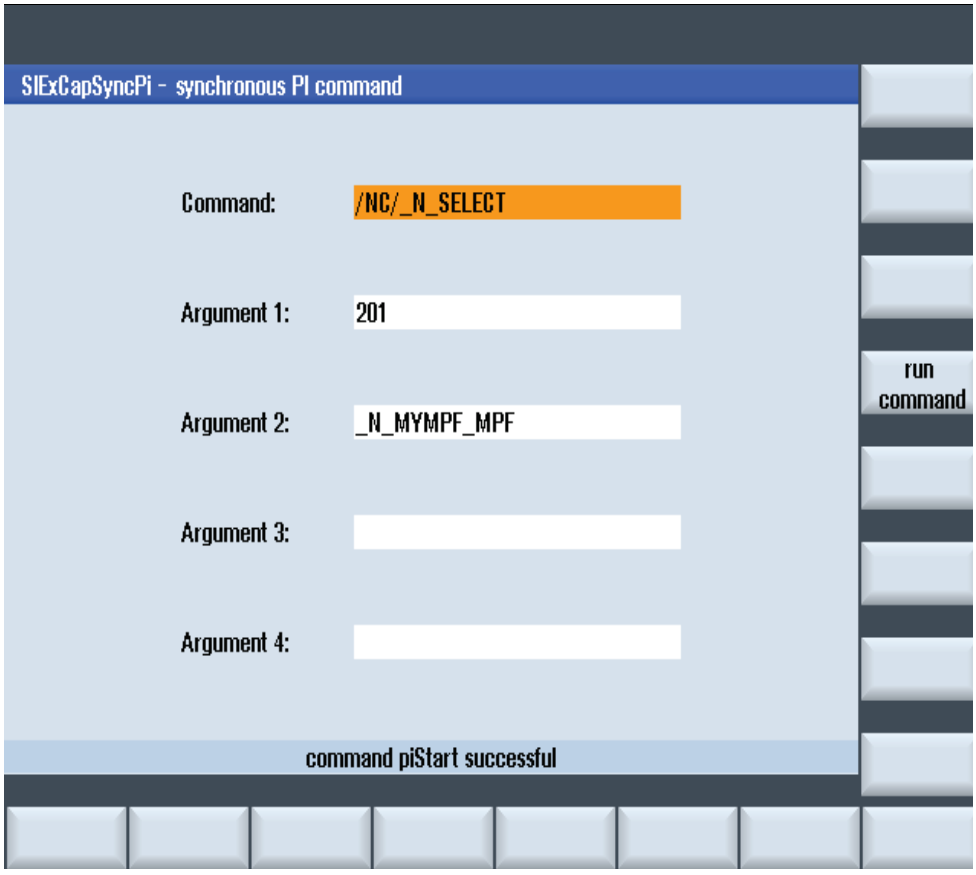


图 5-8:示例 SIExCapSyncPi

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

创建一个 QString 变量，待执行的 PI 命令写入到该变量中。

```
QString szCommand = "/NC/_N_SELECT";
```

第 2 步

创建数据类型为 QString 的一个 QVector。将 PI 命令的参数插入到该变量中。

```
QVector<QString> arguments;
arguments.append("201");
arguments.append("_N_MYMPF_MPF");
```

第 3 步

接着执行 PI 命令`"/NC/_N_SELECT"`。执行结果是选中了第一通道中的零件程序`"_N_MYMPF_MPF"`。该任务的执行状态写入变量 `eError` 中。

```
SlCapErrorEnum eError = m_capServer.piStart(szCommand, arguments);
```

第 4 步

查看 PI 命令的执行状态。

```
if( SL CAP OK  != eError )
{
            // 插入故障处理
}
```

5.3.8 异步启动 PI 命令

概述

下面的示例分步介绍了如何异步启动 PI 命令。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExCapAsyncPi”。该示例程序的结构与 5.3.7 章描述的同步启动 PI 命令的示例相同。

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

作为私有成员变量创建 SlQCapHandle 类的一个对象。该对象用于标识 PI 命令。

```
SlQCapHandle m_capHandlePI;
```

第 2 步

任务的执行结果通过信号通知 CAP 服务。该信号由以下函数（槽）接收。函数的定义在第 7 步 (`executeCompleteSlot`)中变得清晰明确。

```
private slots:
void executeCompleteSlot(SlCapErrorEnum, const QVector<QVariant>&,
                                         const QDateTime&);
```

第 3 步

现在将成员函数（槽）与 CAP 服务发出的信号 `executeComplete`
关联在一起。

```
QObject::connect(&m capServer,
                    SIGNAL(executeComplete(SlCapErrorEnum,
                                           const QVector<QVariant>&,
                                           const QDateTime&)),
                    this,
                    SLOT(executeCompleteSlot(SlCapErrorEnum,
                                           const QVector<QVariant>&,
                                           const QDateTime&)));
```

第 4 步

创建一个 QString 变量，待执行的 PI 命令写入到该变量中。

```
QString szCommand = "/NC/_N_SELECT";
```

第 5 步

创建数据类型为 `QString` 的一个 QVector。将 PI 命令的参数插入到该变量中。

```
QVector<QString> arguments;
arguments.append("201");
arguments.append("_N_MYMPF_MPF");
```

第 6 步

向 CAP 服务传递任务：执行 PI 命令"/NC/_N_SELECT"，以选中第一通道中的零件程序"N_MYMPF_MPF"。

```
SlCapErrorEnum eError = m_capServer.piStartAsync(szCommand,
                                                 arguments,
                                                 &m_capHandlePI);
```

第 7 步

异步调用立即返回。但 PI 命令在调用槽 writeCompleteSlot 后才完成。

```
void SlExCapAsyncPiForm::executeCompleteSlot(SlCapErrorEnum eError,
                                             const QVector<QVariant>&,
                                             const QDateTime&)
{
            if( SL CAP OK  != eError )
            {
                  // 插入故障处理
            }
}
```

5.3.9 同步访问机床数据/GUD

概述

下面的示例分步展示了如何向 CAP 服务声明机床专用数据（机床数据、GUD 等）的同步访问。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExCapSyncMap”。

在本例中，可按下软键“map namespace”声明 NC 机床数据。接着在 LinkItem 下选择访问，按下“read data”读数据。状态栏显示调用成功。

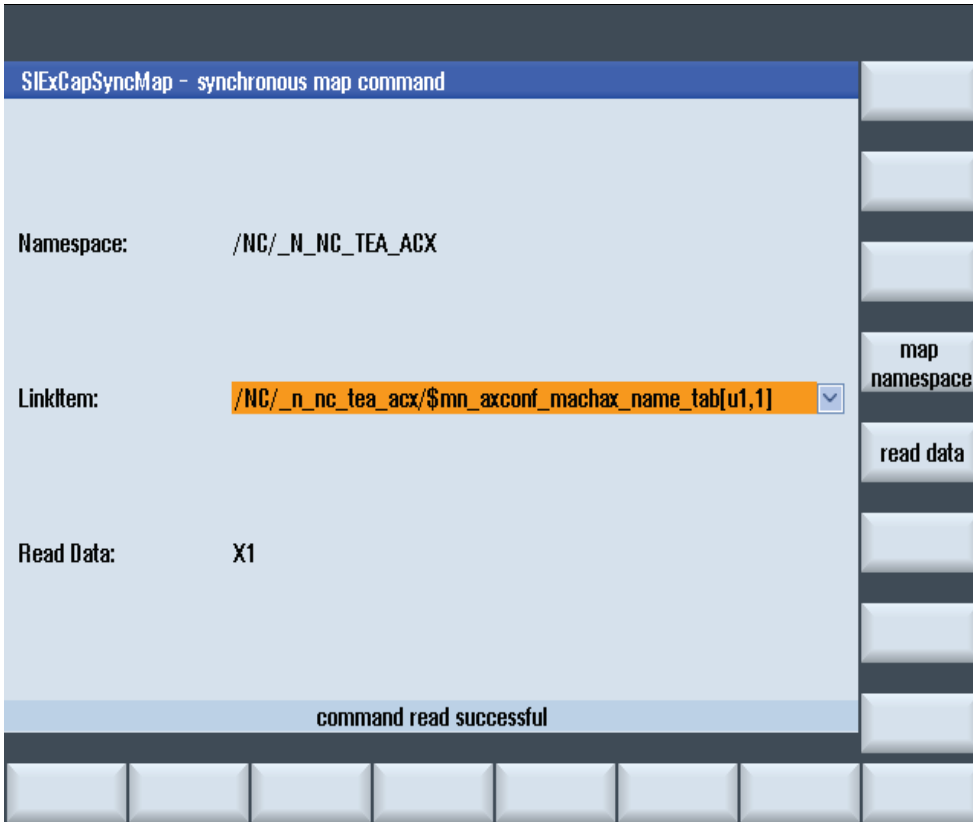


图 5-9:示例 SIExCapSyncMap

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

加入 SIQCapNamespace 类的头文件。

```
#include "slqcapnamespace.h"
```

第 2 步

作为私有成员变量创建 SIQCapNamespace 类的一个对象。

```
SlQCapNamespace m_capNameSpace;
```

第 3 步

在该构造函数中为该对象提供所需的命名空间。本例中为 NC 机床数据。

```
SlExCapSyncMapForm::SlExCapSyncMapForm(QWidget* pParent,
                                       const QString& rszName,
                                       Qt::WFlags f)
                   :SlGfwDialogForm(pParent, rszName, f),
                    m_capNameSpace("/NC/_N_NC_TEA_ACX")
```

第 4 步

激活 NC 机床数据。只有在删除了 SIQCapNamespace 对象后才再次禁止 NC 机床数据。

```
SlCapErrorEnum eError = m_capNameSpace.map();
```

第 5 步

查看 map 命令的执行状态。

```
if( SL CAP OK  != eError )
{
      // 插入故障处理
}
```

第 6 步

如需创建一张包含了所有通过 SlQCapNamespace 来寻址的变量的列表，可使用下述调用。

```
QVariant vLinkItems;
m_capNameSpace.lookup(vLinkItems);
```

第 7 步

分解返回的列表。此时要注意返回的列表中也包含了下级列表。因此必须在循环中再次调用 `toList()`。下级列表的第一个单元是变量的名称。

```
QList<QVariant> varList = vLinkItems.toList();
for( int nIndex = 0 ; nIndex < varList.count() ; nIndex++ )
{
      QList<QVariant> dataList = varList[nIndex].toList();
      QString szVariable       = dataList[0].toString();
      ...
      // szVariable 中提供了有效地址供后续处理
}
```

5.3.10 同步传送零件程序

概述

下面的示例分步展示了如何利用 CAP 服务在 HMI 和 NC 之间同步传送零件程序。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExCapSyncDomainTransfer”。

在本例中，按下软键“create file”可创建一份文件，其中的文本可随意编辑。按下软键“download file to NC”可将该文件传送到指定的预路径中。上传过程与此类似。状态栏显示调用成功。

注

本编程包和 Powerline HMI 编程包的不同之处在于，在传送零件程序时不会删除源文件。用户必须自行确保数据是一致的。

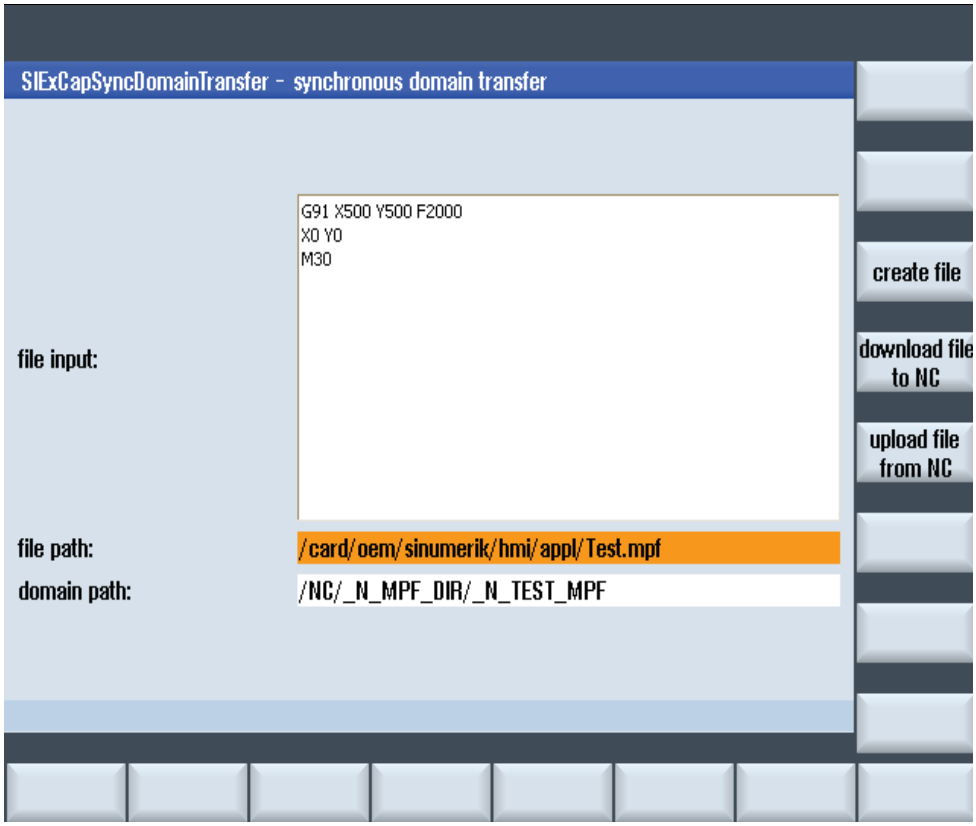


图 5-10:示例 SIExCapSyncDomainTransfer

务必要满足章节 5.3.1 列出的前提条件。

第 1 步

创建一个 QString 变量，其中要填入源文件的路径。

```
QString szFilePath = "/card/oem/sinumerik/hmi/appl/Test.mpf";
```

第 2 步

创建第二个 QString 变量，其中要填入 NC 的域路径（目标路径）。

```
QString szDomainPath = "/NC/_N_MPF_DIR/_N_TEST_MPF";
```

第 3 步

将零件程序“/card/oem/sinumerik/hmi/appl/Test.mpf ”复制到 NC 的路径“/NC/_N_MPF_DIR/_N_TEST_MPF”下。该任务的执行状态写入变量 eError 中。

```
SlCapErrorEnum eError = m_capServer.downloadNc(szFilePath, szDomainPath);
```

第 4 步

查看传送任务的执行状态。

```
if( SL CAP OK  != eError )
{
            // 插入故障处理
}
```

第 5 步

如果希望与上一步相反，将零件程序"/NC/_N_MPF_DIR/_N_TEST_MPF"从 NC 中复制到"/card/oem/sinumerik/hmi/appl/Test.mpf"时，则需要编写以下调用。该任务的执行状态写入变量 eError 中。

```
SlCapErrorEnum eError = m_capServer.uploadNc(szFilePath, szDomainPath);
```

第 6 步

查看传送任务的执行状态。

```
if( SL CAP OK  != eError )
{
      // 插入故障处理
}
```

5.3.11 异步传送零件程序

概述

下面的示例分步展示了如何利用 CAP 服务在 HMI 和 NC 之间异步传送零件程序。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目"SlExCapAsyncDomainTransfer"。该示例程序的结构与 5.3.10 章描述的同步传送零件程序的示例相同。

务必要满足章节 5.3.1 列出的前提条件。

注

本编程包和 Powerline HMI 编程包的不同之处在于，在传送零件程序时不会删除源文件。用户必须自行确保数据是一致的。

第 1 步

作为私有成员变量创建 SlQCapHandle 类的两个对象。其中一个对象用于下载零件程序，另一个对象用于上传零件程序。

除了这两个对象以外，还需要创建 SlQCapHandle 类的另外两个对象。其中第一个标记下载，第二个标记上传。

```
SlQCap m capServerDownload;
SlQCap m_capServerUpload;

SlQCapHandle m_capHandleDownload;
SlQCapHandle m_capHandleUpload;
```

第 2 步

任务的执行结果通过信号通知 CAP 服务。该信号由下面两个函数（槽）接收。这两个函数的定义在第 7 步和第 9 步（executeCompleteSlotDownload 和 executeCompleteSlotUpload）变得清晰明确。

```
private slots:
   void executeCompleteSlotDownload(SlCapErrorEnum, const QVector<QVariant>&,
                                                    const QDateTime&);
   void executeCompleteSlotUpload(SlCapErrorEnum,   const QVector<QVariant>&,
                                                    const QDateTime&);
```

第 3 步

现在将成员函数（槽）与 CAP 服务发出的信号 executeComplete 关联在一起。

```
QObject::connect(&m capServerDownload,
                 SIGNAL(executeComplete(SlCapErrorEnum,
                                        const QVector<QVariant>&,
                                        const QDateTime&)),
                 this,
                 SLOT(executeCompleteSlotDownload(SlCapErrorEnum,
                                        const QVector<QVariant>&,
                                        const QDateTime&)));

QObject::connect(&m_capServerUpload,
                 SIGNAL(executeComplete(SlCapErrorEnum,
                                        const QVector<QVariant>&,
                                        const QDateTime&)),
                 this,
                 SLOT(executeCompleteSlotUpload(SlCapErrorEnum,
                                        const QVector<QVariant>&,
                                        const QDateTime&)));
```

第 4 步

创建一个 QString 变量，其中要填入源文件的路径。

```
QString szFilePath = "/card/oem/sinumerik/hmi/appl/Test.mpf";
```

第 5 步

创建第二个 QString 变量，其中要填入 NC 的域路径（目标路径）。

```
QString szDomainPath = "/NC/_N_MPF_DIR/_N_TEST_MPF";
```

第 6 步

向 CAP 服务传送任务：将零件程序“/card/oem/sinumerik/hmi/appl/Test.mpf”复制到路径“/NC/_N_MPF_DIR/_N_TEST_MPF”下。

```
SlCapErrorEnum eError = m_capServerDownload.downloadNcAsync(
                                            szFilePath,
                                            szDomainPath,
                                            &m_capHandleDownload);
```

第 7 步

异步调用立即返回。但复制任务在调用槽 executeCompleteSlotDownload 后才完成。

```
void SlExCapAsyncDomainTransferForm::executeCompleteSlotDownload(
                                                    SlCapErrorEnum eError,
                                                    const QVector<QVariant>&,
                                                    const QDateTime&)
{
         if( SL_CAP_OK  != eError )
         {
               // 插入故障处理
         }
}
```

第 8 步

如果希望与上一步相反，将零件程序"/NC/_N_MPF_DIR/_N_TEST_MPF"从 NC 中复制到"/card/oem/sinumerik/hmi/appl/Test.mpf"时，则需要编写以下调用。

```
SlCapErrorEnum eError = m_capServerUpload.uploadNcAsync(
                                            szFilePath,
                                            szDomainPath,
                                            &m_capHandleUpload);
```

第 9 步

异步调用立即返回。但复制任务在调用槽 executeCompleteSlotUpload 后才完成。

```
void SlExCapAsyncDomainTransferForm::executeCompleteSlotUpload(
                                                    SlCapErrorEnum eError,
                                                    const QVector<QVariant>&,
                                                    const QDateTime&)
{
         if( SL_CAP_OK  != eError )
         {
               // 插入故障处理
         }
}
```

5.4 SIQCap 引用

5.4.1 定义

概述

访问 NC 和 PLC 上的数据是通过 SIQCap 对象实现的。特别是该对象可以读/写变量，在值变化时发出通知。另外它还可以传送文件，启动 PI 服务。

几乎所有的的调用既可以采取同步方式，也可以采取异步方式。同步调用此时会阻塞 Qt 主线程，直到任务完成。而在异步调用中，SIQCap 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

只存在默认构造函数。无法复制 SIQCap 对象或对其赋值。

变量的映射

在读变量时，结果值以 QVariant 的形式提供。该 QVariant 可以携带不同的数据类型。CAP 服务会选择其中和控制系统内的数据类型最相似的一种类型，对该数据进行完整映射。

同理，在写变量时，数值也要以 QVariant 的形式提供，并要能转化成控制系统内唯一的一种数据类型，否则 CAP 服务会报错。

确定数据类型：

```
QVariant vValue;
QVariant::Type dataTypeVariant = vValue.type();

if ( QVariant::List == dataTypeVariant )
...
if ( QVariant::ByteArray == dataTypeVariant )
...
```

数组访问

如果在变量路径中定义了对多个连续变量的访问（数组访问），CAP 服务会在结果 QVariant 中返回一张 QList<QVariant>。列表元素在数据类型方面和单个变量访问一致。列表的第一个元素（下标=0）即数组的第一个元素。

如果控制系统的数据类型为“signed byte”或“signed byte”，则结果 QVariant 中返回的是 QByteArray，而不是列表。

在写数组时，数据接收到所有可在一个 QVariant 内部传递的数组数据容器中：

- QList<QVariant>
- QStringList
- QByteArray
- QBitArray

连接监控

欲监控连接，要使用以下变量。返回值 SL_CAP_CCS_WORKING (30)表示存在连接。

表 5-8：连接变量

变量	描述
/nc/connect_state	与 NC 的连接监控
/plc/connect_state	与 PLC 的连接监控

注

直到目前还使用变量“/bag/state/opmode”或“/Nck/Configuration/maxNumChannels”用于 NCK 的连接监控，将变量“ib0”用于 PLC。不再建议使用，因为该方法可能不完全满足要求。

所报告数值变化的完整性

通过建立 Hotlink（advise 调用）不能确保会对所有的数值变化进行报告。原因如下：

1. 数值由 NCK 循环读取。如果所读取的值与已经报告的值不同，则会继续报告数值变化。由于这种机制，如果一个值在两个连续的扫描点之间发生了变化并又恢复为原始数值，则不会识别到该变化。

2. 扫描点在时间上不一定是等间隔的。如果实时通讯受阻或者通讯连接过载，NCK/PLC 会抑制继续报告数值变化。

3. 为了减少应用程序收到的数据总量，会尝试在一个通讯协议单元(PDU)中集合尽可能多的数值。

4. 如果客户端出现数据拥堵，则会抑制数值修改。

总结如下：
CAP 服务无法识别到变量的所有数值变化。此外也无法确保所要求的扫描频率，该频率只能视为目标值。但是 CAP 服务仍会尽量向客户端报告它识别出的所有数值变化，只要客户端能作出响应。对于客户端而言此时并不存在实时请求，它必须阻止形成不断增长的反向拥堵。CAP 服务只能保证在经过一定的延时后报告一系列变化后的最终值。

5.4.2 变量路径

用于访问 NC 的变量路径

用于访问 NC 的变量路径可从 BTSS 帮助文件中获取。该帮助文件同样也是 SINUMERIK Operate 编程包文档的组成部分。

表 5-9：变量路径（NC 访问）示例

变量路径	描述
/channel/parameter/r[u1,10]	通道 1 中的 R 参数 10。
/channel/parameter/r[u1,4,#5]	通道 1 中的 R 参数数组，从 R4 到 R8。
/channel/parameter/r[u2,40,42]	通道 2 中的 R 参数数组，从 R40 到 R42。
/channel/geometricAxis/name[u2,3]	通道 2 中第 3 轴的名称。
/channel/geometricAxis/actToolBasePos[u1,3]	通道 1 中第 3 轴的位置。

用于访问 GUD 的变量路径

用于访问 GUD 的变量路径由激活的命名空间路径和 GUD 名称组成，参见章节 5.6“SIQCapNamespace 引用”中的表“命名空间路径”。

GUD 数组在访问 1 中有索引，从 SIQCap 的角度来看始终是一维的，也就是说，在多维数组中必须要计算索引。

示例 1：一维数组

文件 UGUD.DEF

```
DEF NCK INT ARRAY[2]
M17
```

数组访问方式为：

```
ARRAY[0] ➔ /NC/ N NC GD3 ACX/ARRAY[1]
ARRAY[1] ➔ /NC/_N_NC_GD3_ACX/ARRAY[2]
```

示例 2：二维数组

文件 UGUD.DEF

```
DEF CHAN INT ABC[3,3]
M17
```

数组访问方式为：

```
ABC[0,0] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 1]
ABC[0,1] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 2]
ABC[0,2] ➔ /NC/ N CH GD3 ACX/ABC[u1, 3]
ABC[1,0] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 4]
ABC[1,1] ➔ /NC/ N CH GD3 ACX/ABC[u1, 5]
ABC[1,2] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 6]
ABC[2,0] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 7]
ABC[2,1] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 8]
ABC[2,2] ➔ /NC/_N_CH_GD3_ACX/ABC[u1, 9]
```

用于访问 PLC 的变量路径

用于访问 PLC 的变量路径符合 S7 句法。此时既可使用 Simatic，也可使用 IEC 寻址。

表 5-10：PLC 句法

范围	地址 (Simatic)	地址(IEC)	允许的数据类型
输出映像	Ax.y	Qx.y	BOOL
输出映像	ABx	QBx	BYTE, CHAR, STRING, DT
输出映像	AWx	QWx	WORD, INT, DATE, S5TIME, CHAR
输出映像	ADx	QDx	DWORD, DINT, REAL, TIME, TOD
数据块	DBz.DBx.y	DBz.DBx.y	BOOL
数据块	DBz.DBXx.y	DBz.DBXx.y	BOOL
数据块	DBz.DBBx	DBz.DBBx	BYTE, CHAR, STRING, DT
数据块	DBz.DBWx	DBz.DBWx	WORD, INT, DATE, S5TIME, CHAR
数据块	DBz.DBDx	DBz.DBDx	DWORD, DINT, REAL, TIME, TOD
输入映像	Ex.y	Ix.y	BOOL
输入映像	EBx	IBx	BYTE, CHAR, STRING, DT
输入映像	EWx	IWx	WORD, INT, DATE, S5TIME, CHAR
输入映像	EDx	IDx	DWORD, DINT, REAL, TIME, TOD
标记	Mx.y	Mx.y	BOOL
标记	MBx	MBx	BYTE, CHAR, STRING, DT
标记	MWx	MWx	WORD, INT, DATE, S5TIME, CHAR
标记	MDx	MDx	DWORD, DINT, REAL, TIME, TOD
计时器	Tx	Tx	S5TIME
计数器	Zx	Cx	WORD

表格注释：

1. 表中的“x”指数据区中的字节偏移，“y”指字节中的位编号，“z”指数据块。
2. 带有下划线的数据类型是各自的默认数据类型，在寻址时无需说明。因此例如这两种写法 DB2.DBB5:BYTE 和 DB2.DBB5 是相同的。
3. 在访问数组时会使用方括号，例如 DB5.DBW2:[10]（长度为 10 的字数组）。

表 5-11：变量路径（PLC 访问）示例

变量路径	描述
M5.0	从字节偏移 5 开始的标记位 0。
DB5.DW2	数据块 5 中从字节偏移 2 开始的字（16 位）。
DB5.DW2:S5TIME	数据块 5 中从字节偏移 2 开始的字（16 位），作为 S5 时间。
DB8.DBB2:STRING	数据块 8 中从字节偏移 2 开始的 UTF8 字符串。
DB8.DBW2:[10]	数据块 8 中从字节偏移 2 开始的长度为 10 个字的数组。
DB100.DBB1	数据块 100 中从字节偏移 1 开始的字节。
DB100.DBW7:[5]	数据块 100 中从字节偏移 7 开始的长度为 5 个字的数组。
DB2.DBD0:REAL[10]	数据块 2 中从字节偏移 0 开始的长度为 10 个双字（32 位）的数组，格式化为浮点值。

注意以下提示：

- 计时器只能读取。当值不为 0 时，计时器生效。
- 当使用数据类型 CHAR 或 STRING 时，如果是字节访问，就读取 UTF8 字符；如果是字访问，就读取 UTF16 字符。

- STRING 型变量的首个字节/字中为最大长度，第二个字节/字中为实际长度。在写入字符串时，不会修改最大长度。
- 在关系到字节访问（例如：DB99.DBB0:STRING）的 STRING 数据类型上最大字符串长度为 255 个字符。 UTF8 格式中单个字符（例如：字符“µ”）需要两个字节，这样最大字符串长度会减少。
- 只支持一维数组。STRING 型数组的所有元素必须具有相同的最大长度。
- 在进行 CHAR 类型的字段访问时会提供一个字符串，而不是数组。字符串的结尾为第一个识别到的 0 字符。

1:N 访问的变量路径

如果存在 1:N 配置便可以访问特定的控制系统。 可通过前置前缀进行访问：

NC 变量：***@NCU-名称/NC/****变量路径*
PLC 变量: ***@NCU-名称/PLC/****变量路径*

*NCU-名称*符合 *mmc.ini* 中配置的 NCU 名称：

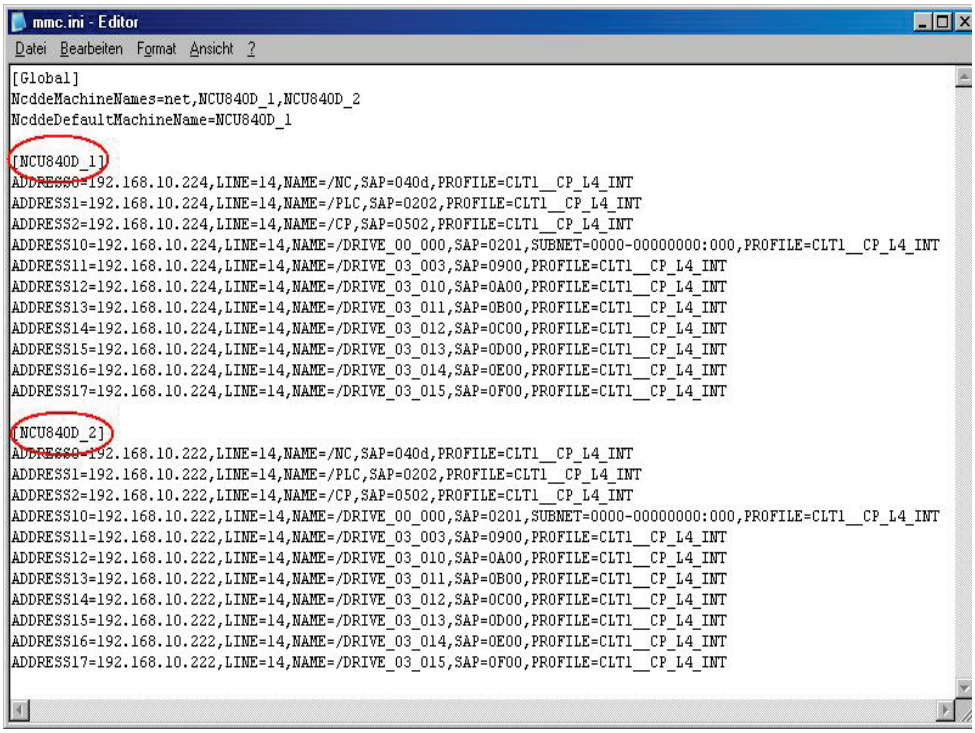


图 5-11：mmc.ini 中的 NCU 名称

表 5-12： 变量路径（1:N 访问）示例

变量路径	描述
@NCU_1/NC/channel/parameter/r[u1,1]	NCU_1 R 参数
@NCU_2/NC/_N_NC_GD3_ACX/MyGud	NCU_2 UGUD 参数
@MYNCU/PLC/DB5.DW2	MYNCU PLC 变量
@NCU_1/NC/_N_SELECT	NCU_1 上的 PI 指令
@NCU_2/NC/_N_MPF_DIR/_N_TEST_MPF	NCU_2 的域路径

用于访问 CTRL energy 的变量路径

以下变量路径可用于访问能量管理(CTRL energy)：

表 5-13：用于访问 CTRL energy 的变量路径

变量	描述
/Hmi/CtrlEnergy/State	**格式：** QVariant / UInt **可用性：** 始终可用 **含义：** 1 ➔ 测量激活 0 ➔ 测量未激活
/Hmi/CtrlEnergy/DeviceNames	**格式：** QVariant / QVariantList<QString> **可用性：** 始终可用 **含义：** 包含了配置设备的名称（不含语言代码）以及设备在 Ctrl energy 表中的名称（total drives, total machine, manual, X1, Y1, …）。
/Hmi/CtrlEnergy/Types	**格式：** QVariant / QVariantList<Int> **可用性：** 始终可用 **含义：** 和 /Hmi/CtrlEnergy/DeviceNames 类似，包含了配置的每个设备的类型代码： 0..31 ➔ 轴 1000 ➔ 手动 1100 ➔ 整个驱动 1200 ➔ SENTRON PAC 或整个机床 1300 ➔整个机床
/Hmi/CtrlEnergy/Power	**格式：** QVariant / QVariantList<double> **可用性：** 始终可用 **含义：** 和 /Hmi/CtrlEnergy/DeviceNames 类似，包含了配置设备的反馈电量。
/Hmi/CtrlEnergy/ActiveEnergy	**格式：** QVariant / QVariantList<double> **可用性：** 在测量激活时可用 **含义：** 和 /Hmi/CtrlEnergy/DeviceNames 类似，包含了配置的每个设备的反馈电量。
/Hmi/CtrlEnergy/ReactiveEnergy	**格式：** QVariant / QVariantList<double> **可用性：** 在测量激活时可用 **含义：** 和 /Hmi/CtrlEnergy/DeviceNames 类似，包含了配置的每个设备的反馈电量。
/Hmi/CtrlEnergy/TotalEnergy	**格式：** QVariant / QVariantList<double> **可用性：** 在测量激活时可用 **含义：** 和 /Hmi/CtrlEnergy/DeviceNames 类似，包含了配置的每个设备的电量总和：馈入电量加上反馈电量。

标准“参数”操作区域中提供的变量路径：

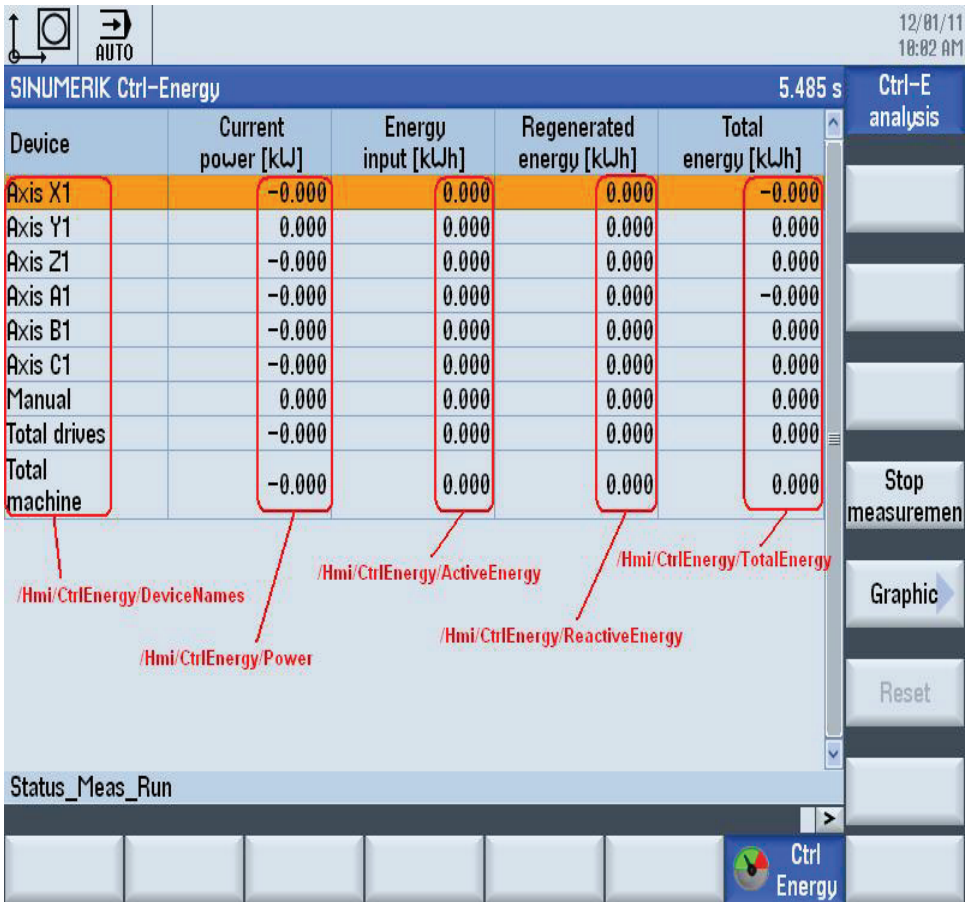


图 5-12:SINUMERIK 中用于访问 CTRL energy 的变量路径

在目录“<安装路径>/oem/sinumerik/hmi/cfg”下创建一份包含以下内容的配置文件“slctrl_energy.ini”，变量路径才能生效：

```
[Misc]
EnableOAInterface=true
```

5.4.3 用于访问配置的函数

查询可访问的服务器(reachableServers)

该函数返回可访问的服务器的名称。

表 5-14：reachableServers

SlCapErrorEnum SlQCap::reachableServers(QList<QString>& lstServers);	
参数	描述
lstServers	一张含可返回的服务器名称的列表。
返回值	函数的执行状态。

查询可访问的 NCU(reachableNcus)

该函数返回一张指出可通过服务器访问的 NCU 的列表。

表 5-15：reachableNcus

SlCapErrorEnum SlQCap::reachableNcus(const QString& strServ, QList<SlCapNcu>& lstNcus);	
参数	描述
strServ	需要查询的服务器。
lstNcus	一张含 NCU 信息的列表。 （参见表 5-13：struct SlCapNcu）
返回值	函数的执行状态。

表 5-16：struct SlCapNcu

struct SlCapNcu
{
QString m_strMachine;
QString m_strVisibleName;
QList<QString> m_lstSubDev;
};	

参数	描述
m_strMachine	NCU 的内部名称。
m_strVisibleName	NCU 名称，供显示。
m_lstSubDev	一张指出通过 NCU 的内部名称可访问的单元的列表。

示例：

```
#include "slqcap.h"

SlQCap capServer;
QList<QString> lstServers;
SlCapErrorEnum eError1 = capServer.reachableServers(lstServers);
if ( SL_CAP_OK != eError1 )
{
            // 错误处理
}
else
{
      for ( int nIndex = 0 ; nIndex < lstServers.count() ; nIndex++ )
      {
         QList<SlCapNcu> lstNCUs;
         SlCapErrorEnum eError2=capServer.reachableNcus(lstServers[nIndex],
                                                                  lstNCUs);
       }
}
```

5.4.4 读取变量

同步读取一个变量(read)

该函数读取一个变量并阻塞当前线程的执行。

该函数也可在辅助线程中使用。

表 5-17：read

SICapErrorEnum SIQCap::read(	**const QString& strVarName, QVariant& vValue, unsigned timeout = SIQCap::standardTimeout(), quint64 uFlags = 0, SICapSupplementInfoType* pSuppl = 0);**
参数	含义
strVarName	变量路径
vValue	返回的变量值。
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
pSuppl	服务器在此处提供更多的读任务信息。 （另见章节 5.4.14“更多的读任务信息”）
返回值	读任务的执行状态。

示例：
另见章节 5.3.2“分步示例”

异步读取一个变量(readAsync)

该函数异步读取一个变量。一旦读任务传送给服务器，函数就返回。如果 readAsync 返回时没有错误，就发送 Qt 信号 readData 结束任务。

异步读取可通过调用 cancel 来取消。同样，只要一识别到相关的 SIQCap 对象被删除，任务也会被取消。

表 5-18：readAsync

SICapErrorEnum SIQCap::readAsync(	**const QString& strVarName, SIQCapHandle* pAsyncId = 0, unsigned timeout = SIQCap::standardTimeout(), quint64 uFlags = 0);**
参数	含义
strVarName	变量路径
pAsyncId	用于标识异步调用的句柄(handle)。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后，readData 会返回，并携带错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

表 5-19：Qt 信号 readData

void readData(SlCapErrorEnum eError, const QVariant& vValue, const SlCapSupplementInfoType& rSuppl);	
参数	**描述**
eError	读任务的执行状态。
vValue	返回的变量值。
rSuppl	服务器在此处提供更多的读任务信息。 （另见章节 5.4.14“更多的读任务信息”）

示例：
另见章节 5.3.3“分步示例”

同步读取多个变量(multiRead)

该函数读取多个变量并阻塞当前线程的执行。这些变量可以位于不同区域中，也可以位于不同通道中。

该函数也可在辅助线程中使用。

表 5-20：multiRead

SlCapErrorEnum SlQCap::multiRead(QVector<SlCapReadSpecType>& vecReadSpec, QVector<SlCapReadResultType>& vecReadResult, unsigned timeout = SlQCap::standardTimeout());	
参数	**含义**
vecReadSpec	要读取的变量的数量，为任意值。
vecReadResult	返回的变量值。每个变量的索引和 vecReadSpec 中的索引一一对应。
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
返回值	读任务的执行状态。所有变量的读任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。

表 5-21：SlCapReadSpecType（选段）

typedef struct SlCapReadSpec { QString m_strVarName; quint64 m_uFlags; } SlCapReadSpecType;	
参数	**含义**
m_strVarName	变量路径
m_uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）

SlCapReadSpecType 可以用不同的方式填入：

```
QVector<SlCapReadSpecType> readSpec(3);
readSpec[0] = "/nck/nck/channel";
readSpec[1] = SlCapReadSpecType("/channel/parameter/r[1]");
readSpec[2] = SlCapReadSpecType("/bag/state/opmode", 0);
```

表 5-22：SICapReadResultType（选段）

```
typedef struct SICapReadResult
{
        QVariant m_vValue;
        SICapErrorEnum m_eError;
        SICapSupplementInfoType m_supplement;
} SICapReadResultType;
```

参数	含义
m_vValue	返回的变量值。
m_eError	单个变量读任务的执行状态。
m_supplement	服务器在此处提供更多的读任务信息。 （另见章节 5.4.14“更多的读任务信息”）

示例：
另见章节 5.3.4“分步示例”

异步读取多个变量(multiReadAsync)

该函数异步读取多个变量。一旦读任务传送给服务器，函数就返回。如果 multiReadAsync 返回时没有错误，就发送 Qt 信号 multiReadData 结束任务。

这些变量可以位于不同区域中，也可以位于不同通道中。

异步读取可通过调用 cancel 来取消。同样，只要一识别到相关的 SlQCap 对象被删除，任务也会被取消。

表 5-23：multiReadAsync

SlCapErrorEnum SlQCap::multiReadAsync(QVector<SlCapReadSpecType>& vecReadSpec, SlQCapHandle* pAsyncId = 0, unsigned timeout = SlQCap::standardTimeout());	
参数	含义
vecReadSpec	要读取的变量的数量，为任意值。
pAsyncId	用于标识异步调用的句柄(handle)。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后，multiReadData 会返回，并携带错误 SL_CAP_CLIENT_TIMEOUT。（另见章节 5.4.13“优化常数”）
返回值	函数的执行状态。所有变量的读任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。

表 5-24：Qt 信号 multiReadData

void multiReadData(SlCapErrorEnum eError, const QVector<SlCapReadResultType>& vecReadResult);	
参数	含义
eError	读任务的执行状态。所有变量的读任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。
vecReadResult	返回的变量值。每个变量的索引和 vecReadSpec 中的索引一一对应。

数据类型 `SlCapReadSpecType` 和 `SlCapReadResultType` 在上文的“同步读取多个变量(multiRead)”中详细说明。

示例：
另见 SINUMERIK Operate 编程包中的示例“SlExCapMultiAsyncReadWrite”

5.4.5 写入变量

同步写入一个变量(write)

该函数写入一个变量并阻塞当前线程的执行。

该函数也可在辅助线程中使用。

表 5-25：read

SICapErrorEnum SIQCap::write(	**const QString& strVarName, const QVariant& vValue, unsigned timeout = SIQCap::standardTimeout(), quint64 uFlags = 0, QDateTime* pTimeStamp = 0);**
参数	含义
strVarName	变量路径
vValue	待写入的值
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
pTimeStamp	返回执行写任务或应答写任务的时间点。无法确定该时间时，此处返回一个无效值，其无效性可通过查询 isValid()确定。
返回值	写任务的执行状态。

示例：
另见章节 5.3.2“分步示例”

异步写入一个变量(writeAsync)

该函数异步写入一个变量。一旦写任务成功传送给服务器，该函数就返回。如果 writeAsync 返回时没有错误，就发送 Qt 信号 writeComplete 结束任务。

异步写入可通过调用 cancel 来取消。同样，只要一识别到相关的 SIQCap 对象被删除，任务也会被取消。

表 5-26：writeAsync

SlCapErrorEnum SlQCap::writeAsync(**const QString& strVarName,** **const QVariant& vValue,** **SlQCapHandle* pAsyncId = 0,** **unsigned timeout = SlQCap::standardTimeout(),** **quint64 uFlags = 0);**	
参数	含义
strVarName	变量路径
vValue	待写入的值
pAsyncId	用于标识异步调用的句柄(handle)。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后，writeComplete 会返回，并携带错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

表 5-27：Qt 信号 writeComplete

void writeComplete(SlCapErrorEnum eError, **const QDateTime& rdateTime);**	
参数	描述
eError	写任务的执行状态。
rdateTime	返回执行写任务或应答写任务的时间点。无法确定该时间时，此处返回一个无效值，其无效性可通过查询 isValid()确定。

示例：
另见章节 5.3.3“分步示例”

同步写入多个变量(multiWrite)

该函数写入多个变量并阻塞当前线程的执行。这些变量可以位于不同区域中，也可以位于不同通道中。

该函数也可在辅助线程中使用。

表 5-28：multiWrite

SlCapErrorEnum SlQCap::multiWrite(**QVector<SlCapWriteSpecType>& vecWriteSpec,** **QVector<SlCapWriteResultType>* vecWriteResult = 0,** **unsigned timeout = SlQCap::standardTimeout());**	
参数	描述
vecWriteSpec	要写入的变量的数量，为任意值。
vecWriteResult	返回执行结果。每个变量的索引和 vecWriteSpec 中的索引一一对应。
Timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
返回值	写任务的执行状态。所有变量的写任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。

表 5-29：SlCapWriteSpecType（选段）

typedef struct SlCapWriteSpec { QString m_strVarName; quint64 m_uFlags; QVariant m_vValue; } SlCapWriteSpecType;	
参数	含义
m_strVarName	变量路径
m_uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
m_vValue	待写入的值

SlCapWriteSpecType 中应填入：

```
QVector<SlCapWriteSpecType> writeSpec(2);
writeSpec[0] = SlCapWriteSpecType("/channel/parameter/r[1]", 1);
writeSpec[1] = SlCapWriteSpecType("/channel/parameter/r[1]", 0.0);
```

表 5-30：SlCapWriteResultType（选段）

typedef struct SlCapWriteResult { SlCapErrorEnum m_eError; QDateTime m_timeStamp; } SlCapWriteResultType;	
参数	含义
m_eError	单个写任务的执行状态。
m_dtTimeStamp	返回执行写任务或应答写任务的时间点。无法确定该时间时，此处返回一个无效值，其无效性可通过查询 isValid()确定。

示例：
另见章节 5.3.4“分步示例”

异步写入多个变量(multiWriteAsync)

该函数异步写入多个变量。一旦写任务成功传送给服务器，该函数就返回。如果 multiWriteAsync 返回时没有错误，就发送 Qt 信号 multiWriteComplete 结束任务。

这些变量可以位于不同区域中，也可以位于不同通道中。

异步写入可通过调用 cancel 来取消。同样，只要一识别到相关的 SlQCap 对象被删除，任务也会被取消。

表 5-31：multiWriteAsync

SlCapErrorEnum SlQCap::multiWriteAsync(QVector<SlCapWriteSpecType>& vecWriteSpec, SlQCapHandle* pAsyncId = 0, unsigned timeout = SlQCap::standardTimeout());	
参数	含义
vecWriteSpec	要写入的变量的数量，为任意值。
pAsyncId	用于标识异步调用的句柄(handle)。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后，multiWriteComplete 会返回，并携带误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
返回值	函数的执行状态。所有变量的写任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。

表 5-32：Qt 信号 multiWriteComplete

void multiWriteComplete(SlCapErrorEnum eError, const QVector<SlCapWriteResultType>&);	
参数	含义
eError	写任务的执行状态。所有变量的写任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。
vecWriteResult	返回执行结果。每个变量的索引和 vecWriteSpec 中的索引一一对应。

数据类型 `SlCapWriteSpecType` 和 `SlCapWriteResultType` 在上文的“同步写入多个变量(multiWrite)”中详细说明。

示例：
另见 SINUMERIK Operate 编程包中的示例“SlExCapMultiAsyncReadWrite”

5.4.6 变量值变化的通知(Hotlink)

建立一个变量的 Hotlink(advise)

建立一个变量值变化时的通知。只要将建立任务传递给服务器，调用就返回。通知是通过 Qt 信号 adviseData 发出的。首次通知提供的是建立时的值。

可能会丢失中间值，而不会丢失最终值（另见
章节 5.4.1“定义”，所报告数值变化的完整性）。

通知可通过调用 unadvise 来取消。同样，只要一识别到相关的 SlQCap 对象被删除，任务也会被取消。

表 5-33：advise

SICapErrorEnum SIQCap::advise(const QString& strVarName, SIQCapHandle& pAdviseId, unsigned requestedUpdate = SIQCap::standardUpdateRate(), double deadBand = 0.0, unsigned timeout = SIQCap::standardTimeout(), quint64 uFlags = 0);	
参数	含义
strVarName	变量路径
pAdviseId	用于标识通知任务的句柄。一旦该句柄被删除就取消任务。
requestedUpdate	循环时间，单位毫秒。该值向 CAP 服务指出以何种频率扫描数值。在目标设备过载时，CAP 服务可能无法保证以目标频率进行扫描。 （另见章节 5.4.13“优化常数”）
deadBand	当新值与旧值差距过小时便不通知值的变化。该抑制只对数值生效。 值大于 0 时，如果新值与旧值的差距没有超过死区，便不会通知值的变化。 值小于 0 时，以该值为单位划分区间。只有新值跃变至一个新的区间后，才通知值的变化。
timeout	发出首个通知的时长限制，单位毫秒。该时间届满后不会再等待后续值，而是发出通知。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

表 5-34：Qt 信号 adviseData

void adviseData(SICapErrorEnum eError, const QVariant& vValue, const SICapSupplementInfoType& rSuppl);	
参数	含义
eError	通知任务的执行状态。
vValue	返回的变量值。
rSuppl	服务器在此处提供更多的读任务信息。 （另见章节 5.4.14“更多的读任务信息”）

示例：
另见章节 5.3.5“分步示例”

建立多个变量的 Hotlink(multiAdvise)

建立多个变量值变化时的通知。只要将建立任务传递给服务器，调用就返回。通知是通过 Qt 信号 multiAdviseData 发出的。首次通知提供的是建立时所有变量的值。可能会丢失中间值，而不会丢失最终值。其他相关信息参见章节“所报告数值变化的完整性”。

通知可通过调用 unadvise 来取消。同样，只要一识别到相关的 SlQCap 对象被删除，任务也会被取消。

表 5-35：multiAdvise

SlCapErrorEnum SlQCap::multiAdvise(QVector<SlCapAdviseSpecType>& vecAdviseSpec, SlQCapHandle& pAdviseId, unsigned timeout = SlQCap::standardTimeout());	
参数	含义
vecAdviseSpec	通知任务的数量，为意值。
pAdviseId	用于标识通知任务的句柄。一旦该句柄被删除就取消任务。
timeout	发出首个通知的时长限制，单位毫秒。该时间届满后不会再等待后续值，而是发出通知。（另见章节 5.4.13“优化常数”）
返回值	函数的执行状态。

表 5-36：SlCapAdviseSpecType（选段）

typedef struct SlCapAdviseSpec { QString m_strVarName; unsigned m_requestedUpdate; double m_deadBand; quint64 m_uFlags; } SlCapAdviseSpecType;	
参数	含义
m_strVarName	变量路径
m_requestedUpdate	循环时间，单位毫秒。该值向 CAP 服务指出以何种频率扫描数值。在目标设备过载时，CAP 服务可能无法保证以目标频率进行扫描。 （另见章节 5.4.13“优化常数”）
m_deadBand	当新值与旧值差距过小时便不通知值的变化。该抑制只对数值生效。 值>0:如果新值与旧值的差距没有超过死区，便不会通知值的变化。 值<0:以该值为单位设置区间。只有新值跃变至一个新的区间后，才通知值的变化。
m_uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）

SlCapAdviseSpecType 可以用不同的方式填入：

```
QVector<SlCapAdviseSpecType> aviseSpec(2);
aviseSpec[0] = "/nck/nck/channel";
aviseSpec[1] = SlCapAdviseSpecType("/channel/parameter/r[1]");
```

表 5-37：Qt 信号 multiAdviseData

void multiAdviseData(SlCapErrorEnum eError, const QVector<SlCapAdviseResultType>& vecAdviseResult);	
参数	含义
eError	通知任务的执行状态。所有变量的通知任务都返回相同值时，此处显示成功执行。如果有几个变量的任务返回不同值，此处显示 SL_CAP_DIFFERENT_ERRORS。
vecAdviseResult	返回的变量值。每个变量的索引和 vecAdviseSpec 中的索引一一对应。

表 5-38：SlCapAdviseResultType（选段）

typedef struct SlCapAdviseResult { QVariant m_vValue; SlCapErrorEnum m_eError; SlCapSupplementInfoType m_supplement; } SlCapAdviseResultType;	
参数	含义
m_vValue	返回的变量值。
m_eError	单个通知任务的执行状态。
m_supplement	服务器在此处提供更多的读任务信息。 （另见章节 5.4.14“更多的读任务信息”）

示例：
另见 SINUMERIK Operate 编程包中的示例“SlExCapMultiDataChange”

删除 Hotlink(unadvise)

该函数删除用 advise 或 multiAdvise 建立的通知。

表 5-39：unadvise

SlCapErrorEnum SlQCap::unadvise(SlQCapHandle& adviseId);	
参数	含义
adviseId	在执行 advise 或 multiAdvise 时返回的句柄。
返回值	函数的执行状态。

示例：
另见章节 5.3.5“分步示例”
另见 SINUMERIK Operate 编程包中的示例“SlExCapMultiDataChange”

暂停 Hotlink(suspend)

该函数暂停用 advise 或 multiAdvise 建立的通知。

表 5-40：suspend

SICapErrorEnum SIQCap::suspend(SIQCapHandle& adviseId);	
参数	含义
adviseId	在执行 advise 或 multiAdvise 时返回的句柄。
返回值	函数的执行状态。

恢复 Hotlink(resume)

该函数恢复用 advise 或 multiAdvise 建立、然后用 suspend 暂停的通知。

表 5-41：resume

SICapErrorEnum SIQCap::resume(SIQCapHandle& adviseId, unsigned timeout = SIQCap::standardTimeout());	
参数	含义
adviseId	在执行 advise 或 multiAdvise 时返回的句柄。
timeout	调用时长限制，单位毫秒。 （另见章节 5.4.13“优化常数”）
返回值	函数的执行状态。

5.4.7 跨域传送文件

同步下载文件(download, downloadNc)

该函数将文件（比如零件程序）同步传送到 NC 的一个域中。一旦传送任务完成，函数便返回。可以在 Qt 信号 progress 中查看传送进度（百分比值）。

在执行函数 downloadNc 时，NC 需要使用的元信息一同加入到数据流中。

表 5-42：download/downloadNc

SICapErrorEnum SIQCap::download(const QString& filePath, const QString& domainPath, unsigned timeout = SIQCap::extendedTimeout(), quint64 uFlags = 0); SICapErrorEnum SIQCap::downloadNc(const QString& filePath, const QString& domainPath, unsigned timeout = SIQCap::extendedTimeout(), quint64 uFlags = 0);	
参数	含义
filePath	源文件的路径名称（即源）。
domainPath	NC 域的路径名称（即目标）。
timeout	调用时长限制，单位毫秒。该时间届满后，传送任务会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	传送任务的执行状态。

表 5-43：Qt 信号 progress

void progress(int nProgress, const QDateTime& dateTime);	
参数	含义
nProgress	执行进度，百分比值。
dateTime	生成 Qt 信号的时间。

示例：
另见章节 5.3.10“分步示例”

异步下载文件(downloadAsync, downloadNcAsync)

该函数将文件（比如零件程序）异步传送到 NC 的一个域中。只要将传送任务传递给服务器，函数就返回。在执行期间，函数会将执行进度（百分比值）传送给 Qt 信号 progress。任务结束后发送 Qt 信号 executeComplete。

在执行函数 downloadNcAsync 时，NC 需要使用的元信息一同加入到数据流中。

任务可通过调用 cancel 来取消。同样，只要一识别到相关的 SIQCap 对象被删除，任务也会被取消。

表 5-44：downloadAsync/downloadNcAsync

```
SICapErrorEnum SIQCap::downloadAsync(
                    const QString& filePath,
                    const QString& domain,
                    SIQCapHandle* domainId = 0,
                    unsigned timeout = SIQCap::extendedTimeout(),
                    quint64 uFlags = 0);

SICapErrorEnum SIQCap::downloadNcAsync(
                    const QString& filePath,
                    const QString& domain,
                    SIQCapHandle* domainId = 0,
                    unsigned timeout = SIQCap::extendedTimeout(),
                    quint64 uFlags = 0);
```

参数	含义
filePath	源文件的路径名称（即源）。
domain	NC 域的路径名称（即目标）。
domainId	用于标识传送任务的句柄。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后发送 Qt 信号 executeComplete，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

表 5-45：Qt 信号 executeComplete

```
void executeComplete(   SICapErrorEnum eError,
                        const QVector<QVariant>& outArgs,
                        const QDateTime& dateTime);
```

参数	含义
eError	传送任务的执行状态。
outArgs	空
dateTime	返回任务结束或被应答的时间。

Qt 信号 `progress` 在上文“同步下载文件(download, downloadNc)”中详细说明。

示例：
另见章节 5.3.11“分步示例”

同步上传文件(upload, uploadNc)

该函数将 NC 文件上的一个域（比如零件程序）同步传送到 HMI 的一份文件中。一旦传送任务完成，函数便返回。可以在 Qt 信号 progress 中查看传送进度（百分比值）。
在执行函数 uploadNc 时，NC 中随附提供的元信息会被从数据流中删除。

表 5-46：upload/uploadNc

```
SICapErrorEnum SIQCap::upload(
                              const QString& filePath,
                              const QString& domain,
                              unsigned timeout = SIQCap::extendedTimeout(),
                              quint64 uFlags = 0);

SICapErrorEnum SIQCap::uploadNc(
                              const QString& filePath,
                              const QString& domain,
                              unsigned timeout = SIQCap::extendedTimeout(),
                              quint64 uFlags = 0);
```

参数	含义
filePat	目标文件的路径名称（即目标）。
domain	NC 域的路径名称（即源）。
timeout	调用时长限制，单位毫秒。该时间届满后，传送任务会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	传送任务的执行状态。

Qt 信号 progress 在上文“同步下载文件(download, downloadNc)”中详细说明。

示例：
另见章节 5.3.10“分步示例”

异步上传文件(uploadAsync, uploadNcAsync)

该函数将 NC 文件上的一个域（比如零件程序）异步传送到 HMI 的一份文件中。只要将传送任务传递给服务器，函数就返回。在执行期间，函数会将执行进度（百分比值）传送给 Qt 信号 progress。任务结束后发送 Qt 信号 executeComplete。

在执行函数 uploadNc 时，NC 中随附提供的元信息会被从数据流中删除。

任务可通过调用 cancel 来取消。同样，只要一识别到相关的 SIQCap 对象被删除，任务也会被取消。

表 5-47：uploadAsync/uploadNcAsync

```
SICapErrorEnum SIQCap::uploadAsync(
                    const QString& filePath,
                    const QString& domain,
                    SIQCapHandle* pAsyncId = 0,
                    unsigned timeout = SIQCap::extendedTimeout(),
                    quint64 uFlags = 0);

SICapErrorEnum SIQCap::uploadNcAsync(
                    const QString& filePath,
                    const QString& domain,
                    SIQCapHandle* pAsyncId = 0,
                    unsigned timeout = SIQCap::extendedTimeout(),
                    quint64 uFlags = 0);
```

参数	含义
filePath	目标文件的路径名称（即目标）。
domain	NC 域的路径名称（即源）。
pAsyncId	用于标识传送任务的句柄。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后发送 Qt 信号 executeComplete，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

Qt 信号 `progress` 在上文“同步下载文件(download, downloadNc)”中详细说明。

Qt 信号 `executeComplete` 在上文“异步下载文件(downloadAsync, downloadNcAsync)”中详细说明。

示例：
另见章节 5.3.11“分步示例”

5.4.8 管道式跨域传送

同步下载管道(download, downloadNc)

该函数将通过 writePipe(Async)生成的数据流同步传送到 NC 的一个域中。一旦传送任务完成，函数便返回。如果赋值了参数 fileSize，便可以在 Qt 信号 progress 中查看传送进度（百分比值）。

这两个函数会在 Qt 信号 sendPipeData 中发出请求，以启动向管道写入数据的任务。writePipe(Async)不能在 Qt 信号 sendPipeData 前调用。

在执行函数 downloadNc 时，NC 需要使用的元信息一同加入到数据流中。

表 5-48：download/downloadNc

SlCapErrorEnum SlQCap::download(const QString& domainPath, unsigned fileSize = SL_CAP_UNKNOWN_SIZE, unsigned timeout = SlQCap::extendedTimeout(), quint64 uFlags = 0); **SlCapErrorEnum SlQCap::downloadNc(const QString& domainPath, unsigned fileSize = SL_CAP_UNKNOWN_SIZE, const QDateTime* pLastChange = 0, unsigned timeout = SlQCap::extendedTimeout(), quint64 uFlags = 0);**	
参数	含义
domainPath	NC 域的路径名称（即目标）。
fileSize	需要传送的字节的数量。该数量用于 Qt 信号 progress 计算出执行百分比。指定了 SL_CAP_UNKNOWN_SIZE 时，会封锁该信号。
pLastChange	NC 元信息上次修改的时间。如果该参数为 0，则使用当前时间。
timeout	调用时长限制，单位毫秒。该时间届满后，传送任务会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	传送任务的执行状态。

表 5-49：Qt 信号 progress

void progress(int nProgress, const QDateTime& dateTime);	
参数	含义
nProgress	执行进度，百分比值。
dateTime	生成 Qt 信号的时间。

表 5-50：Qt 信号 sendPipeData

void sendPipeData(const QDateTime& dateTime);	
参数	含义
dateTime	返回在服务器中启动任务的时间。

示例：
另见 SINUMERIK Operate 编程包中的示例“SlExCapSyncPipedDomainTransfer”

异步下载管道(downloadAsync, downloadNcAsync)

该函数将通过 writePipe/writePipeAsync 生成的数据流异步传送到 NC 的一个域中。只要将传送任务传递给服务器，函数就返回。如果赋值了参数 fileSize，在执行期间，函数会将执行进度（百分比值）传送给 Qt 信号 progress。任务结束后发送 Qt 信号 executeComplete。

这两个函数会在 Qt 信号 sendPipeData 中发出请求，以启动向管道写入数据的任务。writePipe/writePipeAsync 不能在 Qt 信号 sendPipeData 前调用。
在执行函数 downloadNc 时，NC 需要使用的元信息一同加入到数据流中。

任务可通过调用 cancel 来取消。同样，只要一识别到相关的 SlQCap 对象被删除，任务也会被取消。

表 5-51：downloadAsync/downloadNcAsync

```
SICapErrorEnum SIQCap::downloadAsync(
                        const QString& domain,
                        SIQCapHandle* domainId = 0,
                        unsigned fileSize = SL_CAP_UNKNOWN_SIZE,
                        unsigned timeout = SIQCap::extendedTimeout(),
                        quint64 uFlags = 0);

SICapErrorEnum SIQCap::downloadNcAsync(
                        const QString& domain,
                        SIQCapHandle* domainId = 0,
                        unsigned fileSize = SL_CAP_UNKNOWN_SIZE,
                        const QDateTime* pLastChange = 0,
                        unsigned timeout = SIQCap::extendedTimeout(),
                        quint64 uFlags = 0);
```

参数	含义
domain	NC 域的路径名称（即目标）。
domainId	用于标识传送任务的句柄。一旦该句柄被删除就取消任务。
fileSize	需要传送的字节的数量。该数量用于 Qt 信号 progress 计算出执行百分比。指定了 SL_CAP_UNKNOWN_SIZE 时，会封锁该信号。
pLastChange	NC 元信息上次修改的时间。如果该参数为 0，则使用当前时间。
timeout	调用时长限制，单位毫秒。该时间届满后发送 Qt 信号 completeExecute，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	传送任务的执行状态。

表 5-52：Qt 信号 executeComplete

```
void executeComplete(   SICapErrorEnum eError,
                        const QVector<QVariant>& outArgs,
                        const QDateTime& dateTime);
```

参数	含义
eError	传送任务的执行状态。
outArgs	为空。
dateTime	返回任务结束或被应答的时间。

Qt 信号 `progress` 和 `sendPipeData` 在上文“同步下载管道(download, downloadNc)”中详细说明。

示例：
另见 SINUMERIK Operate 编程包中的示例“SIExCapAsyncPipedDomainTransfer”

同步上传管道(upload, uploadNc)

该函数通过 Qt 信号 pipeData 将一个域同步传送给客户端。一旦传送任务完成，函数便返回。如果域中包含了长度信息，函数会将执行进度（百分比值）传送给 Qt 信号 progress。Qt 信号 pipeData 用于传送域数据。

在执行函数 uploadNc 时，NC 中随附提供的元信息会被从数据流中删除。

表 5-53：upload/uploadNc

SICapErrorEnum SIQCap::upload(const QString& domain, unsigned timeout = SIQCap::extendedTimeout(), quint64 uFlags = 0); SICapErrorEnum SIQCap::uploadNc(const QString& domain, unsigned timeout = SIQCap::extendedTimeout(), quint64 uFlags = 0);

参数	含义
domain	NC 域的路径名称（即源）。
timeout	调用时长限制，单位毫秒。该时间届满后，传送任务会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。（另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	传送任务的执行状态。

表 5-54：Qt 信号 pipeData

void pipeData(const QByteArray& data, bool bEof);

参数	含义
数据	提供的管道数据。
bEof	如果是最后一次提供数据，则此处返回 true。

Qt 信号 `progress` 在上文“同步下载管道(download, downloadNc)”中详细说明。

示例：
另见 SINUMERIK Operate 编程包中的示例“SIExCapSyncPipedDomainTransfer”

异步上传管道(uploadAsync, uploadNcAsync)

该函数通过 Qt 信号 pipeData 将一个域异步传送给客户端。只要将传送任务传递给服务器，函数就返回。如果域中包含了长度信息，函数会将执行进度（百分比值）传送给 Qt 信号 progress。Qt 信号 pipeData 用于传送域数据。任务结束后发送 Qt 信号 executeComplete。

在执行函数 uploadNcAsync 时，NC 中随附提供的元信息会被从数据流中删除。

任务可通过调用 cancel 来取消。同样，只要一识别到相关的 SIQCap 对象被删除，任务也会被取消。

表 5-55：uploadAsync/uploadNcAsync

SICapErrorEnum SIQCap::uploadAsync(const QString& domain, SIQCapHandle* pAsyncId = 0, unsigned timeout = SIQCap::extendedTimeout(), quint64 uFlags = 0); SICapErrorEnum SIQCap::uploadNcAsync(const QString& domain, SIQCapHandle* pAsyncId = 0, unsigned timeout = SIQCap::extendedTimeout(), quint64 uFlags = 0);	
参数	描述
domain	NC 域的路径名称（即源）。
pAsyncId	用于标识传送任务的句柄。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后发送 Qt 信号 completeExecute，但含错误 SL_CAP_CLIENT_TIMEOUT。这种情况下，传送任务被取消。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	传送任务的执行状态。

Qt 信号 `progress` 在上文“同步下载管道(download, downloadNc)”中详细说明。

Qt 信号 `executeComplete` 在上文“异步下载管道(downloadAsync, downloadNcAsync)”中详细说明。

Qt 信号 `pipeData` 在上文“同步上传管道(upload, uploadNc)”中详细说明。

示例：
另见 SINUMERIK Operate 编程包中的示例“SIExCapAsyncPipedDomainTransfer”

同步写入管道(writePipe)

该函数向“piped download”的一个管道中写入数据。客户端一直被挂起，直到 CAP 服务中的数据缓冲器被传送任务清空一半。

在同一个启动传送任务的 SIQCap 对象中启动 writePipe。

表 5-56：writePipe

SICapErrorEnum SIQCap::writePipe(SIQCapHandle& domainId, const void* data, int dataByteLength, bool bEof, unsigned timeout = SL_CAP_NO_TIMEOUT);	
参数	含义
domainId	标识对应的下载任务。
数据	需要传送其数据的缓冲器。
dataByteLength	需要传送的字节的数量。
bEof	指明在数据传送完成后结束传送任务。
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
返回值	写任务的执行状态。

示例：
另见 SINUMERIK Operate 编程包中的示例“SIExCapSyncPipedDomainTransfer”

异步写入管道(writePipeAsync)

该函数向“piped download”的一个管道中写入数据。只要将传送任务传递给服务器，函数就返回。

一旦 CAP 服务的数据缓冲器被传送任务清空一半，便通过 Qt 信号 pipeWriteReturn 向客户端通知该状态。在最后一次调用 writePipeAsync 时（参数 bEof = true），不会发出该信号。同样在下载取消后也不再发送该信号。

异步写入管道的任务不能通过调用 cancel 来取消。只有其中的 pipe transfer(up/download)任务能通过调用 cancel 来取消。但该调用也会使异步写入管道任务结束。

在同一个启动传送任务的 SIQCap 对象中启动 writePipeAsync。

表 5-57：writePipeAsync

SICapErrorEnum SIQCap::writePipeAsync(SIQCapHandle& domainId, const void* data, int dataByteLength, bool bEof, unsigned timeout = SL_CAP_NO_TIMEOUT);	
参数	含义
domainId	标识对应的下载任务。
数据	需要传送其数据的缓冲器。
dataByteLength	需要传送的字节的数量。
bEof	指明在数据传送完成后结束传送任务。
timeout	调用时长限制，单位毫秒。该时间届满后暂时发送 Qt 信号 pipeWriteReturn，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
返回值	函数的执行状态。

表 5-58：Qt 信号 pipeWriteReturn

void pipeWriteReturn(SlCapErrorEnum eError, const QDateTime& dateTime);	
参数	含义
eError	写任务的执行状态。
dateTime	缓冲器被清空一半的时间。

示例：
另见 SINUMERIK Operate 编程包中的示例“SlExCapAsyncPipedDomainTransfer”

5.4.9 PI 命令

同步执行 PI 启动命令(piStart,仅限 NCK)

该函数同步执行一个 PI 启动命令。它只能执行 NCK 的 PI 服务。

表 5-59：piStart

```
SlCapErrorEnum SlQCap::piStart(
                    const QString& strCommand,
                    const QVector<QString>& arguments,
                              unsigned timeout = SlQCap::standardTimeout(),
                              quint64 uFlags = 0,
                    QDateTime* pTimeStamp = 0);

SlCapErrorEnum SlQCap::piStart(
                    const QString& strCommand,
                    const char* arg01,
                    const char* arg02 = 0,
                    const char* arg03 = 0,
                    ...
                    const char* arg20 = 0);
```

参数	含义
strCommand	PI 命令。
arguments	PI 参数。
arg01, arg02, ...	PI 参数
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
pTimeStamp	执行 PI 启动命令的时间点。
返回值	PI 命令的执行状态。

示例：
另见章节 5.3.7“分步示例”

异步执行 PI 启动命令(piStartAsync,仅限 NCK)

该函数异步执行一个 PI 启动命令。它只能
执行 NCK 的 PI 服务。只要 PI 命令成功传递给服务器，函数就返回。PI 命令结束后发送 Qt 信号 executeComplete。

异步执行 PI 启动命令可通过调用 cancel 来取消。同样，只要一识别到相关的 SIQCap 对象被删除，任务也会被取消。

表 5-60：piStartAsync

SICapErrorEnum SIQCap::piStartAsync(const QString& strCommand, const QVector<QString>& arguments, SIQCapHandle* pAsyncId = 0, unsigned timeout = SIQCap::standardTimeout(), quint64 uFlags = 0);	
参数	含义
strCommand	PI 命令。
arguments	PI 参数。
pAsyncId	用于标识 PI 任务的句柄。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后暂时发送 Qt 信号 executeComplete，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

表 5-61：Qt 信号 executeComplete

void executeComplete(SICapErrorEnum eError, const QVector<QVariant>& outArgs, const QDateTime& dateTime);	
参数	含义
eError	PI 命令的执行状态。
outArgs	为空。
dateTime	返回任务结束或被应答的时间。

示例：
另见章节 5.3.8“分步示例”

5.4.10 访问信号上下文

查询 SlQCap 对象(signallingQCapObject)

该函数访问正在发送 Qt 信号的 SlQCap 对象。

表 5-62：signallingQCapObject

static SlQCap::SlQCap* signallingQCapObject();	
参数	含义
返回值	指向正在发送 Qt 信号的 SlQCap 对象的指针。如果当前没有发送 Qt 信号，则为 0。

查询 SlQCapHandle(signallingQCapHandle)

该函数返回正在发送 Qt 信号的 SlQCapHandle 对象。

表 5-63：signallingQCapHandle

static SlQCapHandle SlQCap::signallingQCapHandle();	
参数	含义
返回值	标识正在发送 Qt 信号的对象的句柄。如果当前没有发送任何 Qt 信号，则该句柄无效。

5.4.11 修改异步调用

取消命令(cancel)

cancel 取消正在执行的异步任务，结束通知。该命令在发送任务后立即返回。在命令返回后不再发送任何 Qt 信号。

通常情况下正在执行的任务是无法取消的，比如已经发送给 NCU 的读任务或写任务。此时 cancel 只是封锁相关 Qt 信号的发送。

一些比较耗时的 NCU 任务，比如文件传送任务会在下一个可能的位置上取消。 因此客户端通常需要假定，在任务中断和 cancel 返回之间还间隔一段较短的时间。

表 5-64：cancel

SlCapErrorEnum SlQCap::cancel(SlQCapHandle pAsyncId);	
参数	含义
pAsyncId	用于标识需要取消的任务的句柄。
返回值	取消任务的执行状态。

5.4.12 用于修改数据访问的标志位

概述

通过参数 uFlags 可以修改函数的正常执行过程。设置了多个标志位时，这些标志位进行逻辑或运算。

```
SlCapErrorEnum eError = m_capServer.multiWrite(...,
                            SL_CAP_FLAG_LOW_PRIORITY | SL_CAP_FLAG_TOGETHER);
```

处理优先级

CAP 服务有 3 种处理优先级：

SL_CAP_FLAG_LOW_PRIORITY
低处理优先级。该标志位最好应用于后台循环处理的任务。在执行 advise 任务时，一旦扫描到第一个值，CAP 服务便默认使用该优先级。

SL_CAP_FLAG_HIGH_PRIORITY
高处理优先级。该标志位仅供机床制造商为达到特定生产目的（比如换刀）使用。

SL_CAP_FLAG_DEFAULT_PRIORITY
默认处理优先级。该标志位(flag)用于用户互动和画面结构改造。CAP 服务将所有读/写任务都都归入该默认优先级执行。同样，advise 任务中
第一个值的访问也以该优先级进行。

注意下列边界条件：

1. 由于高处理优先级的任务会取代低处理优先级的任务，因此循环任务(Hotlink)只能使用低处理优先级 SL_CAP_FLAG_LOW_PRIORITY。

2. 处理优先级只在 CAP 服务内部生效。已在途处理的低优先级任务无法再提升其优先级。

注

标志位 SL_CAP_FLAG_DEFAULT_PRIORITY（值 = 0）不需要显式设置。没有设置可选参数 uFlag 时，会自动设置该优先级。

重合数据的渗透性

偶尔多个单个数据会相互参考，并必须同时读取或写入。但 CAP 服务不能确保多变量调用中的变量都在一个任务中发送。尤其是当通讯负载较高时，它反而会尝试优化其通流能力。该属性可通过以下标志位设置：

SL_CAP_FLAG_TOGETHER
在多变量调用的多个或所有变量上设置该标志位后，可以强制使选中的变量在一个任务中传送，并且在一个任务中返回结果。

设置标志位 SL_CAP_FLAG_TOGETHER 后还需注意几个附加边界条件，如不满足将拒绝整个任务。边界条件如下：

1. 变量的单位必须统一（PLC、NC 或驱动）。
   --> 错误：SL_CAP_DIFFERENT_ADDR

2. 对于 NCK 变量而言，通道专用数据只允许在一个通道中寻址，驱动专用数据只允许在一个驱动中寻址。
   --> 错误：SL_CAP_DIFFERENT_TARGET

3. 所有变量必须以相同的访问级访问。
   --> 错误：SL_CAP_DIFFERENT_PRIORITY

4. 任务和结果不得超过 PDU 的容量。
   --> 错误：SL_CAP_DOES_NOT_FIT_INTO_ONE_PDU

注

一个 PDU(protocol data unit)中的可用空间取决于多个终端设备（NC、PLC、驱动）参数，因此无法给出一般性的数据。参考值如下：
读取任务： 至少 16 个变量的空间
写入任务： 至少 8 个变量的空间
净数据量： 约 190 字节

其他标志位

SL_CAP_FLAG_NO_PROGRESS
在跨域传送文件时，该标志位可封锁 Qt 信号 progress 的输出。

5.4.13 优化常数

概述

在编程 Timeouts、UpdateRates 和 FileTransferPuffer 时，建议只访问下文指出的几个常数。

时间监控常数

```
unsigned SlQCap::standardTimeout();
```
默认值，但跨域传送除外。该 Timeout 值应用于快速访问。其会考虑到可能的通讯延时、连接建立等（20000 毫秒）。

```
unsigned SlQCap::extendedTimeout();
```
跨域传送的默认值。该 Timeout 值应用于耗时传送，比如：文件传送。其会考虑到可能的通讯延时、连接建立等（200000 毫秒）。

```
SL_CAP_NO_TIMEOUT;
```
无时间监控。

扫描频率

`unsigned SlQCap::standardUpdateRate();`
默认值，适用于所有 advise 调用。该扫描频率用于数值很少发生变化、但一旦变化就会影响显示的变量。此类变量比如有机床数据、运行模式等（200 毫秒）。

`unsigned SlQCap::highUpdateRate();`
该扫描频率用于数值经常变化、操作人员需要随时查看其变化的变量。此类变量比如有轴位置（50 毫秒）。

`unsigned SlQCap::lowUpdateRate();`
该扫描频率用于在时间方面对值的采集要求不严格的变量（1000 毫秒）。

`unsigned SlQCap::veryLowUpdateRate();`
该扫描频率用于在时间方面对值的采集要求不严格的变量（10000 毫秒）。

文件传送缓冲器的容量

指在下载时可以缓存的字节数。缓冲器被清空一半后，CAP 服务会向客户端请求新的数据。客户端因此发送占一半缓冲器容量的数据。

`unsigned SlQCap::standardDownloadBuffer();`
下载时的默认缓冲器容量（10000 个字节）。

5.4.14 更多的读任务信息

在执行读任务或通知任务时很少进行处理的信息会集合到下面所述的程序段中。

注

下文未提及的参数可能是还没有实现，也可能是和 SINUMERIK Operate 编程包的应用不相关。

表 5-65：struct SICapSupplementInfo （选段）

```
struct SICapSupplementInfo
{
        SICapQualityEnum m_eDataQuality;
        QDateTime m_timeStamp;
        QVariant m_vServerInfo;
        unsigned m_uUpdateRate;
        quint64 m_uSequenceNumber;
} SICapSupplementInfoType;
```

参数	含义
m_eDataQuality	参见“提示”。
m_timeStamp	在 CAP 服务中最后接收到所读取数据的时间点。
m_vServerInfo	由 CAP 服务添加的附加信息。只与读取和设置 PLC 时钟 (clock)相关。
m_uUpdateRate	参见“提示”。
m_uSequenceNumber	通过此序列号可区分任务被执行的前后顺序。序列号覆盖了 CAP 服务中的所有任务。这样任务的序列号虽然是单向递增的，但并不连续。

5.5 SIQCapHandle 引用

5.5.1 定义

概述

在所有异步调用或触发 Qt 信号的调用中，都会返回 SIQCapHandle 类的一个对象。该类的各个对象可以标识出对应的任务。

删除 SIQCapHandle 的对象会取消对应的任务。

可以复制并赋值该类的对象以及检查对象是否相等。

5.5.2 函数

检查格式的有效性(valid)

该函数检查 SIQCapHandle 的格式是否正确，比如检查句柄是否是由一个异步任务返回的。由缺省构造函数生成的句柄无效。

表 5-66：valid

bool SIQCapHandle::valid() const;	
参数	**描述**
返回值	有效性。

查询执行状态(busy)

该函数查询句柄对应的异步任务是正在执行还是已经结束。对于 Hotlink 通知任务而言，只要任务存在（即使处于状态“暂停”中），该函数就返回 true。

表 5-67：busy

bool SIQCapHandle::busy() const;	
参数	**描述**
返回值	执行状态。

查询 Hotlink 的状态(suspended)

该函数查询当前 Hotlink 是否处于“暂停”状态中。

表 5-68：suspended

bool SIQCapHandle::suspended() const;	
参数	**描述**
返回值	Hotlink 的状态。

释放资源，取消异步任务(reset)

该函数用于取消一个异步任务或 Hotlink，和删除句柄的方式一样。该异步任务占用的资源被重新释放。

表 5-69：复位

void SlQCapHandle::reset();	
参数	含义
返回值	无

5.6 SIQCapNamespace 引用

5.6.1 定义

概述

SIQCapNamespace 类的对象可以向 CAP 服务声明机床专用的数据访问（诸如机床数据、全局用户数据 GUD 等）。这些声明在 SIQCapNamespace 对象的整个寿命内都可用。

状态消息和异步任务的执行结果是通过
Qt 信号发送的。

注

变量在 SIQCapNamespace 对象激活后便可被访问，在删除该对象后被删除。

状态

SIQCapNamespace 有三种状态：

1. SL_CAP_BUSY
   表示命名空间正在创建。该状态是刚刚激活时的状态或是在 CAP 服务发现内部描述文件不再是最新文件时的状态。在该状态中不明确是否存在了命名空间变量，是否可访问变量。

2. SL_CAP_OK
   命名空间成功创建，可用。

3. <错误>
   创建命名空间出错。一旦出错，便无法再退出该状态。

5.6.2 构造函数

创建一个 SIQCapNamespace 对象并初始化该对象。接着可以用“map”或“mapAsync”激活命名空间。

表 5-70：valid

SIQCapNamespace(const QString& strPath);	
参数	含义
strPath	命名空间的路径。

可以用下列路径：

表 5-71：命名空间的路径

strPath	含义
"/NC/_N_NC_TEA_ACX"	NC 全局机床数据。
"/NC/_N_CH_TEA_ACX"	通道专用机床数据。
"/NC/_N_AX_TEA_ACX"	轴专用机床数据。
"/NC/_N_NC_SEA_ACX"	NC 全局设定数据。
"/NC/_N_CH_SEA_ACX"	通道专用设定数据。
"/NC/_N_AX_SEA_ACX"	轴专用设定数据。
"/NC/_N_CH_GD?_ACX"	通道专用用户数据(GUD)。 （要使用索引 1 到 9，而不是问号。）
"/NC/_N_NC_GD?_ACX"	NC 全局用户数据（GUD） （要使用索引 1 到 9，而不是问号。）

注

显示机床数据的变量值位于文件“\siemens\sinumerik\hmi\cfg\HMI_MD.INI”中。这些变量值无法通过 SIQCapNamespace 对象访问。

示例：
另见章节 5.3.9“分步示例”
另见 SINUMERIK Operate 编程包中的示例“SIExCapAsyncMap”

5.6.3 激活命名空间

同步激活命名空间(map)

该函数同步激活命名空间。在函数返回后可立即访问命名空间的变量。

注

map 任务的执行时长取决于变量数量。变量较少时可能只要持续几毫秒，变量较多时可能要持续几秒。

表 5-72：map

SICapErrorEnum SIQCapNamespace::map(unsigned uTimeout = SICap::extendedTimeout(), quint64 uFlags = 0);	
参数	**含义**
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	map 任务的执行状态。

表 5-73：Qt 信号 stateChange

void stateChange(SICapErrorEnum eState);	
参数	**含义**
eState	SIQCapNamespace 对象的新状态。

示例：
另见章节 5.3.9“分步示例”

异步激活命名空间(mapAsync)

该函数异步激活命名空间。在调用后 SIQCapNamespace 对象进入状态 SL_CAP_BUSY。只有在过渡到状态 SL_CAP_OK 后，才可以访问命名空间的变量。

命名空间的状态变化通过 Qt 信号“stateChanged”通知或通过“state”查询。

表 5-74：mapAsync

SICapErrorEnum SIQCapNamespace::mapAsync(unsigned uTimeout = SICap::extendedTimeout(), quint64 uFlags = 0);	
参数	含义
timeout	调用时长限制，单位毫秒。该时间届满后，命名空间过渡到状态 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

Qt 信号 `stateChange` 在上文“同步激活命名空间(map)”中详细说明。

示例：
另见 SINUMERIK Operate 编程包中的示例“SIExCapAsyncMap”

5.6.4 状态查询

同步查询变量数据(lookup)

该函数返回一张包含命名空间变量的列表。

表 5-75：lookup

SICapErrorEnum SIQCapNamespace::lookup(QVariant& vValue, unsigned timeout = SICap::standardTimeout(), uint64 uFlags = 0);	
参数	含义
vValue	QVariant 的列表。列表中的每个元素以 QVariant 下级列表的形式描述了一个变量。第一个元素包含了变量名称。
timeout	调用时长限制，单位毫秒。该时间届满后，函数会返回，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	查询任务的执行状态。

示例：
另见章节 5.3.9“分步示例”

异步查询变量数据(lookupAsync)

该函数返回一张包含命名空间变量的列表。被调用的数据通过 Qt 信号 lookupData 传送。

表 5-76：lookupAsync

SlCapErrorEnum SlQCapNamespace::lookupAsync(SlQCapHandle* pAsyncId = 0, unsigned timeout = SlCap::standardTimeout(), quint64 uFlags = 0);	
参数	含义
pAsyncId	用于标识异步调用的句柄(handle)。一旦该句柄被删除就取消任务。
timeout	调用时长限制，单位毫秒。该时间届满后发送 Qt 信号 lookupData，但含错误 SL_CAP_CLIENT_TIMEOUT。 （另见章节 5.4.13“优化常数”）
uFlags	修改正常的执行过程。 （另见章节 5.4.12“用于修改数据访问的标志位”）
返回值	函数的执行状态。

表 5-77：Qt 信号 lookupData

void lookupData(SlCapErrorEnum eError, const QVariant& vData);	
参数	含义
eError	查询任务的执行状态。
vData	QVariant 的列表。列表中的每个元素以 QVariant 下级列表的形式描述了一个变量。第一个元素包含了变量名称。

示例：
另见 SINUMERIK Operate 编程包中的示例“SlExCapAsyncMap”

查询命名空间的状态(state)

该函数返回命名空间的状态。

表 5-78：状态

SlCapErrorEnum SlQCapNamespace::state() const;	
参数	含义
返回值	命名空间的状态。

5.7 常见问题(FAQ)

5.7.1 一般常见问题（接口、SIQCap.....）

是在栈上还是在堆上创建 SIQCap 对象？

SIQCap 类的对象只有少数的实例数据，在创建和删除时费时较短。因此，无论是在栈上还是在堆上都可以创建 SIQCap 对象。

此处要建议您删除不再需要使用的 SIQCap 对象，以便再次释放通讯资源。

涉及显示数据（轴位置、参数等）时，我们建议您作为窗体类的私有成员变量(SIGfwDialogForm)创建一个 SIQCap 对象。摧毁窗体后，该 SIQCap 对象也一并被删除。

可以有多个 SIQCap 对象吗？

在一个类内可以创建多个 SIQCap 对象。只要使用 Hotlink，就不可避免地要创建多个 SIQCap 对象，这是因为一个 SIQCap 对象始终只能同时处理一个任务，所以每个 Hotlink 都需要一个自己的 SIQCap 对象。

需要调用 disconnect 命令断开信号与槽的关联吗？

断开信号与槽之间的关联是由 Qt 在后台执行的。您无需显式编程 disconnect。

如何确定当前访问级？

当前访问级最好不要用 OPI 变量“/Nck/Configuration/accessLevel”来确定，而是用 GUI Framework 在窗体类 `SlGfwDialogForm` 中提供的虚拟函数 `onAccessLevelChanged` 来确定。通过改写该函数可以对当前访问级的改变作出所需响应。

如果需要确定其他时间点上的访问级，应使用函数 `SlGfwHmiDialog::accessLevel()`。

5.7.2 有关 Hotlink 的常见问题

每个 Hotlink 都需要一个 SIQCap 对象吗？

是的。由于一个 SIQCap 对象始终只能同时处理一个任务，因此每个 Hotlink 都需要一个自己的 SIQCap 对象。另外，还需要创建一个 SIQCapHandle 对象以便暂停 Hotlink。

何时需要暂停或删除 Hotlink？

不再需要使用某个 Hotlink 时，最好暂停("suspend")或删除该 Hotlink("unadvise")。不可见窗体的 Hotlink 只会浪费通讯资源。

GUI Framework 为此在窗体类 SIGfwDialogForm 中提供两个虚拟函数：一个是函数 attachedToScreen，在窗体确实在屏幕中可见时调用。另一个是函数 detachedFromScreen，在窗体不再可见时调用。因此最好在函数 attachedToScreen 中启动或激活 Hotlink，在函数 detachedFromScreen 中删除或暂停 Hotlink。

需要暂停一个 Hotlink 然后再恢复该 Hotlink 时，是使用“suspend/resume”还是使用“unadvise/advise”？

这两种方法没有哪一种在时间上占优势。但在创建了多个变量的 Hotlink 时，最好还是使用比较简单的“suspend/resume”，因为在该方法中作为参数只需要创建 SIQCapHandle 对象。如果使用“unadvise”，之后还需要整个重建 Hotlink。

5.7.3 有关机床数据/GUD 的常见问题

我已经成功声明了机床数据或 GUD，但还是无法访问变量。这是为什么？

机床数据或 GUD 的声明需要使用执行“map”或“mapAsync”的一个 SIQCapNamespace 对象。一旦该 SIQCapNamespace 对象被删除，会一并禁止命名空间，之后便无法通过一个 SIQCap 对象访问数据。

在下面的例子中，SIQCapNamespace 对象只存在于 If 块内，因此命名空间也只在该块内激活。在该块范围之外命名空间被禁止，因此无法访问它的变量。

```
if ( true == bNoMapping )
{
      SlQCapNamespace capNameSpace("/NC/_N_NC_TEA_ACX");
      capNameSpace.map();

      // 生效
      m capServer.read("/NC/ n nc tea acx/$mn tu name[u1,1]",...);
}

// 返回错误
m_capServer.read("/NC/_n_nc_tea_acx/$mn_tu_name[u1,1]",...);
```

因此我们建议将 SIQCapNamespace 对象和 SIQCap 对象一起作为私有成员变量创建。这样可以保证命名空间的寿命等于类的寿命。

一个 SIQCapNamespace 可以用于多个 map 任务吗？

不可以。SIQCapNamespace 对象的命名空间在创建（构造函数）时一次性确定。没有任何 set 函数可事后修改命名空间。

需要声明多个空间时，比如：声明 GUD4 和 GUD8，便需要多个 SIQCapNamespace 对象。

如何访问显示机床数据？

显示机床数据的变量值位于文件“\siemens\sinumerik\hmi\cfg\HMI_MD.INI”中。这些变量值无法通过 SIQCapNamespace 对象访问。

建议使用 slhmiutilitieslib.lib 中的类来读/写显示机床数据的单个变量。

```
#include "slhmisettingsqt.h"

QVariant vReadData;
SlHmiSettingsQt setting;
long lRetVal = setting.readConfigEntry("hmi_md/BTSS-Settings",
                                        "$MM_DISPLAY_RESOLUTION", vReadData);
QString szRetVal = vReadData.toString();
```

6

6 访问报警和事件

本章主要内容

本章主要介绍 SINUMERIK Operate 中隶属于服务组件的报警与事件服务（Alarm & Event）。报警与事件服务提供相关函数来处理来自 PLC、NCK（带驱动报警）和 HMI 的报警和信息。
它可提供 HMI 以及与 HMI 相连的其中一个 NCU（NCK 和 PLC）的全部报警和信息。报警和信息包括：

- HMI 报警
- NCK 报警，包括驱动报警（只要报警由 NCK 进行转发）
- 来自 NCK 的零件程序信息
- PLC 的诊断缓冲器报警和信息(FC10)
- 带有条件分析结果的 Alarm_S(Q)报警 (SFC17/18, PDiag, HiGraph, S7-Graph)

报警与事件服务为接口提供以下函数：

- 设置、应答和删除报警
- 封锁并释放报警源
- 筛选报警
- 选择报警可提供的属性。

所有要采集报警和事件（例如用于显示或后续处理以及转发至 MES）的应用程序，在下文中都称为客户端(Client)。

6.1 概述

6.1.1 引言及术语解释

报警和事件

报警表示一个**时间段**，在此期间进程(NCK/PLC/HMI)中存在待处理的异常或临界状态。与此相反，事件表示一个特定的**时间点**，在此时刻报警出现、消失或被用户应答（出现事件、消失事件或应答事件）。依据事件的不同，一条报警会有不同的状态（参见章节 6.2“报警状态机”），一个事件表示状态转变。图 6-1 展示了两者的关联。

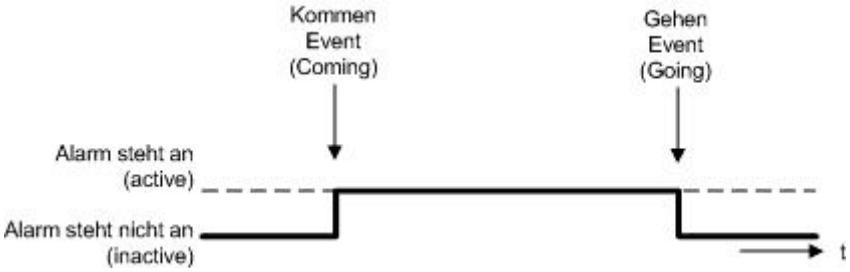


图 6-1:报警和事件

客户端（Client），服务（Service）和源（Source）

报警与事件服务不仅为实现报警客户端提供函数，也为实现报警源提供函数。
报警客户端一般是一个应用程序，用来监视报警及其状态、参数、时间戳等。它通常和报警与事件服务进行通讯。报警与事件服务则与系统中不同的报警源通讯，即 HMI 报警、PLC 报警和 NCK 报警（含驱动报警）。
因此如果一个 HMI 应用程序需要输出自己的报警，必须实现一个报警源。
报警源即可用于 NCK 报警的模拟，也可用于 PLC 报警的模拟。
下面各小节将介绍不同报警类型下客户端、服务与源之间的信号流。

报警源

报警与事件服务可提供来自 NC 中不同报警源的报警。报警源分为以下几种：

表 6-1：报警与事件服务中默认的报警源名称(Source URL)

来源	源 URL
HMI	/HMI
NCK	/NCK
NCK 通道 （只用于零件程序信息）	/NCK/Channel#<通道号>/Partprogram
PLC 诊断缓冲器	/PLC
PLC PMC 报警 (Alarm_S(Q):PDiag, S7-Graph 和 HiGraph)	/PLC/PMC

报警特性(Attribute)

每个报警除了向客户端提供标识报警的报警号外，还提供其他数据，例如状态、时间戳或进程关联值。这些数据被称作特性(Attribute)，分为标准特性和附加特性(Attribute)。标准特性(Attribute)对于所有的报警类别都一样（参见报警类别），附加特性(Attribute)可依类别的不同而不同。
特性(Attribute)还根据其来源加以区分：其一方面可直接来自报警源（例如时间戳、进程关联值），另一方面还可以来源于配置（例如优先级、显示用前景色）。
附加特性可以通过配置来添加。为使客户端可以使用这些特性，报警与事件服务提供一个 Query 函数，它可返回特性的 ID、数据类型和文字说明。另外客户端还可以选择每条报警向它返回的附加特性。

报警类别

并非所有的报警都支持同样的特性(Attribute)构成：比如零件程序信息就没有进程关联值。为区分报警特性(Attribute)，将报警划分为了不同的类别(Categories)，同一类别的所有报警支持相同的特性(Attribute)构成。
取决于配置的不同，报警与事件服务会返回不同类别的报警。因此该服务提供了接口以查询报警类别及其特性构成。
此处 OEM 可以为现有报警类别添加新的特性或者创建新的类别。

HMI 报警

HMI 报警是 HMI 自己产生的报警，即此时 HMI 应用程序担任报警源(Alarm-Source)的角色。为此报警与事件服务为每个客户端提供一个单独的对象(`SlAeQEventSource`)。

6.1.2 工作原理和类模型

单独事件：SlAeEvent 类

在报警与事件服务中，一个报警或事件始终是作为 SlAeEvent 类的一个实例表示的。
下文说明的订阅(Subscripition)以 SlAeEvent 类一个实例的形式向客户端发送报警与事件。有一些是以 SlAeEvents 类的实例列表的形式发送的。

每个报警或事件除了向客户端提供标识报警的报警号外，还提供其他数据，例如状态、时间戳或进程关联值。这些数据被称作特性(Attribute)，分为标准特性和附加特性(Attribute)。所有报警类别的标准特性都一样，都有自己的访问函数。而不同报警类别的附加特性有所不同，因此有统一的访问函数。

! **重要提示**

在报警与事件服务中，一个报警或事件始终是作为 SlAeEvent 类的一个实例表示的。

订阅

客户端对报警的访问（例如用于报警显示）是通过所谓的订阅机制(Subscription)进行的，其中报警与事件服务为每个客户端提供单独的订阅管理。报警的状态发生变化时，客户端可通过 Qt 信号得知自己的订阅。
借助订阅每个客户端都可以选择报警的类型、每条报警包含的数据内容，而不会影响到其他客户端的选择（筛选）。也就是说，每个客户端可以从自己的角度定义报警。
有 3 种（类）订阅方式用于报警显示：

- 报警列表的订阅（`SlAeQAlarmPtrList` 类）
- 事件列表的订阅（`SlAeQEventPtrList` 类）
- 单独事件的订阅（`SlAeQEventSink` 类）

这些订阅方式共用一个函数集。但也有一些函数只供特定订阅方式使用。

注

不能设置多个客户端共用一个订阅。

报警列表的订阅（SlAeQAlarmPtrList 类）

该类向客户端返回一张当前所有待处理报警的列表，其中指出了报警的状态、报警的所有特性以及某语言的报警文本。

一旦报警状态或其特性发生改变，该类就加入一条新条目（报警出现）、修改一条旧条目（比如：报警被应答）或删除旧条目（报警消失）。只要没有激活筛选，列表条目的数量就始终等于当前待处理报警的数量。

列表的每次变化都会通过一个 Qt 信号通知给客户端。
另见章节 6.3.2“显示当前所有待处理的报警（报警列表）”中的示例。

事件列表的订阅（SlAeQEventPtrList 类）

该类向客户端返回一张事件的列表，即报警的状态变化列表。每次状态变化（出现、消失、应答）都会向列表加入一条新条目。但列表有容量限制，因此根据 FIFO 原理（先进先出），新事件会删除最旧的条目。

原则上，在报警与事件服务中新事件是按照其出现的时间先后顺序在列表中排列的。但是由于不同报警源的不同的运行时间或不同的计时器，列表中的事件不一定是按照时间排序的。目前事件列表还不能和报警列表一样按照其他条件排序。

列表的每次变化都会通过一个 Qt 信号通知给客户端。

单独事件的订阅（SIAeQEventSink 类）

SIAeQEventSink 类是应用在 Qt 主线程中的，可向一个客户端应用程序通知报警的状态变化（即事件）。通知是依据 Qt 的信号与槽机制进行的，每发生一个事件便发出一个信号。这些事件可以是报警的出现事件、消失事件或应答事件，取决于报警的状态机。刷新事件也可以用这种方式通知。对于同一个报警源而言，比如：NCU_1 中 PLC 的诊断缓冲器，信号的发送顺序始终和该报警源中事件的发生顺序一致。由于操作系统中偶尔进行的线程/进程切换，不同报警源发出的信号无法可靠地实现同步，尤其是这些报警源的事件在时间上非常靠近时。

客户端槽 `newEvent` 的执行没有时间限制。在执行期间出现新的事件时，这些事件依次排入 Qt 主线程的消息队列(FIFO)中。
该类的其他所有函数都以同步方式调用，在调用期间会阻塞 Qt 主线程的执行。
另见章节 6.3.3“显示当前所有发生的单独事件（EventSink）”中的示例。

创建报警源：SIAeQEventSource

除了显示功能外，报警与事件服务还提供了一些函数，用于设置 HMI 报警、将 HMI 报警从客户端中删除。通过这些函数您可以在客户端中实现自定义的报警源。
另外，通过这些函数您还可以模拟 NCK 报警和 PLC 报警。
另见章节 6.3.4“创建 HMI 报警（EventSource）”中的示例。

6.2 报警状态机

报警状态机描述的是报警可能出现的状态变化。

6.2.1 需应答的报警

需应答的报警可以有几下几种状态：

CGA: **C**oming + **G**oing + **A**cknowledged (Reset State)
（出现 + 消失 + 应答）

C: **C**oming
（出现）

CA: **C**oming + **A**cknowledged
（出现 + 应答）

CG: **C**oming + **G**oing (but not Acknowledged)
（出现 + 消失，但未应答）

需要应答的报警比如有来自 S7-PDiag、S7-Graph 和 S7-HiGraph 的报错信息以及来自 PLC 的 Alarm_SQ 报警(SFC17)。此外取决于“出现事件”的类型(SLAE_ALARM_COMING_TOACK)，HMI 报警也可能需要应答。

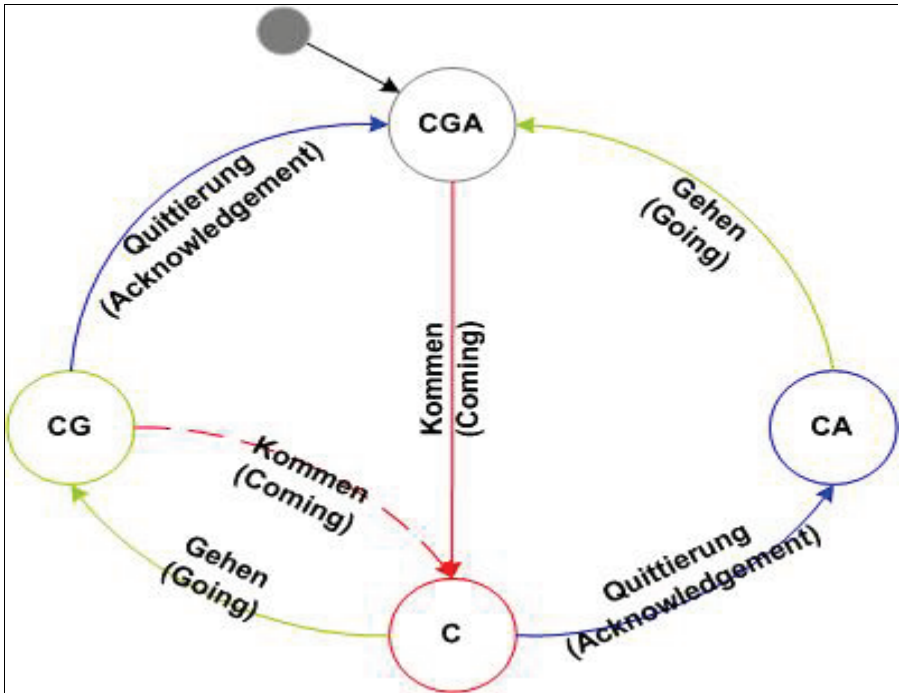


图 6-2:需应答的报警的状态机

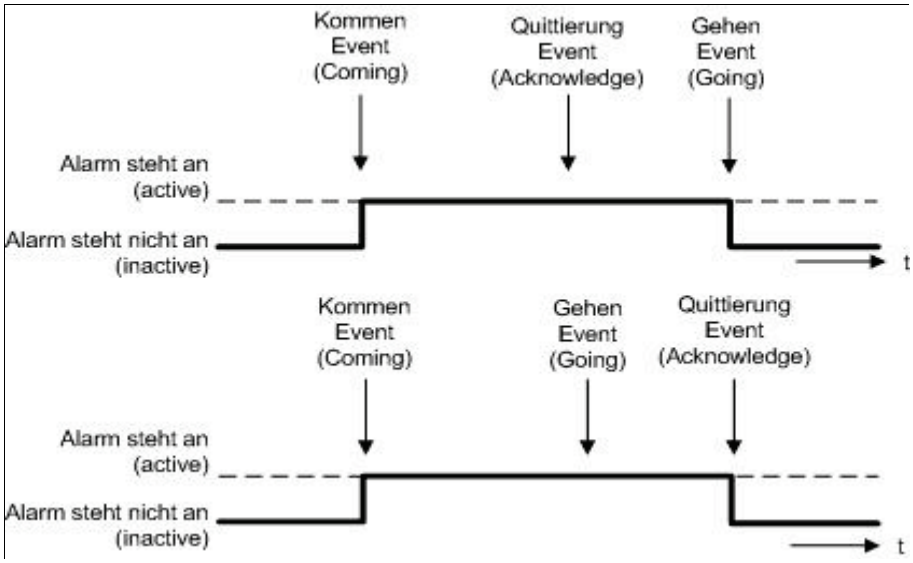


图 6-3:需应答的报警包含的事件（含应答请求的报警）

注

目前还不支持单独应答出现事件和消失事件。在报警与事件服务中只有出现事件是强制需要应答的。

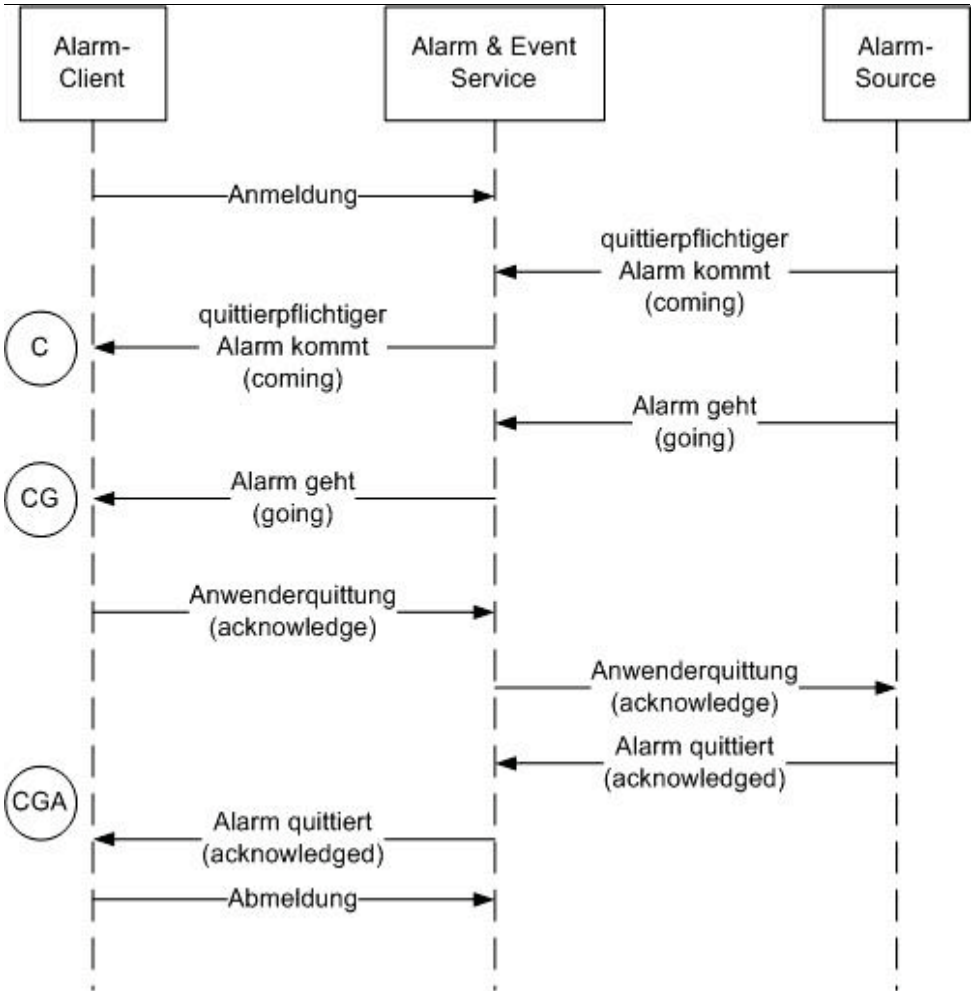


图 6-4:要应答的报警的信号流： 示例 1

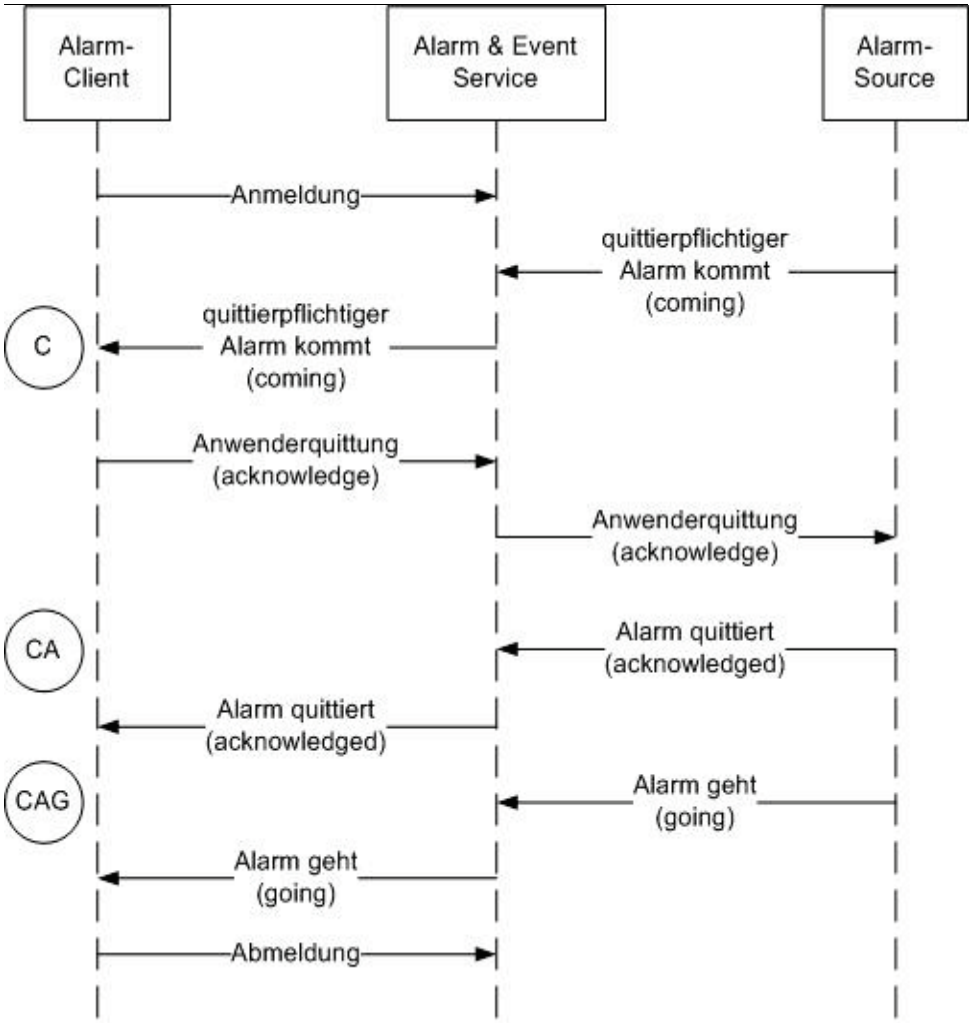


图 6-5:要应答的报警的信号流： 示例 2

6.2.2 无需应答的报警

无需应答报警只有两种状态，此类报警可自动应答：

CGA: **C**oming + **G**oing + **A**cknowledged (Reset-State)
（出现 + 消失 + 应答）

CA: **C**oming + **A**cknowledged
（出现 + 应答）

无需应答的报警比如有：所有 NCK 报警（含驱动报警）、零件程序信息（MSG 命令）、PLC 的 DB2 报警/信息、来自 S7-PDiag、S7-Graph 和 S7-HiGraph 的运行信息以及来自 PLC 的 Alarm_S 报警(SFC18) 。此外取决于“出现事件”的类型 (SLAE_ALARM_COMING)，HMI 报警也可能无需应答。

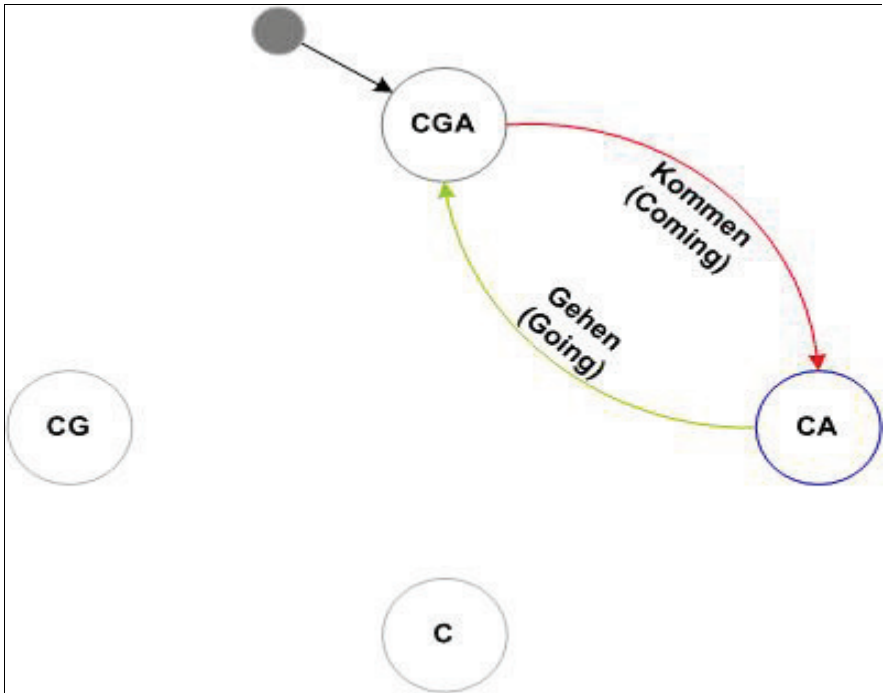


图 6-6:无需应答的报警的状态机

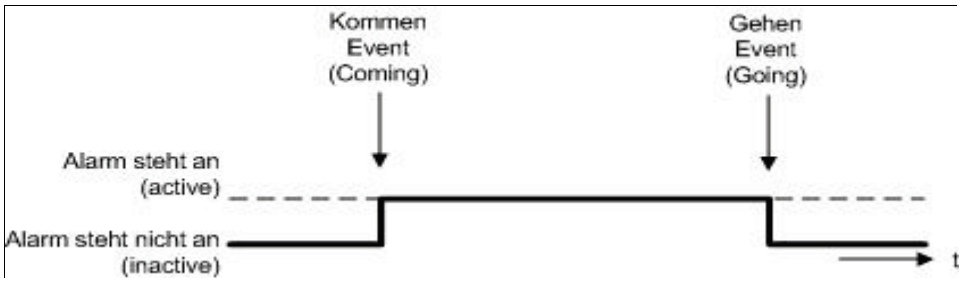


图 6-7:无需应答的报警包含的事件（不含应答请求的报警）

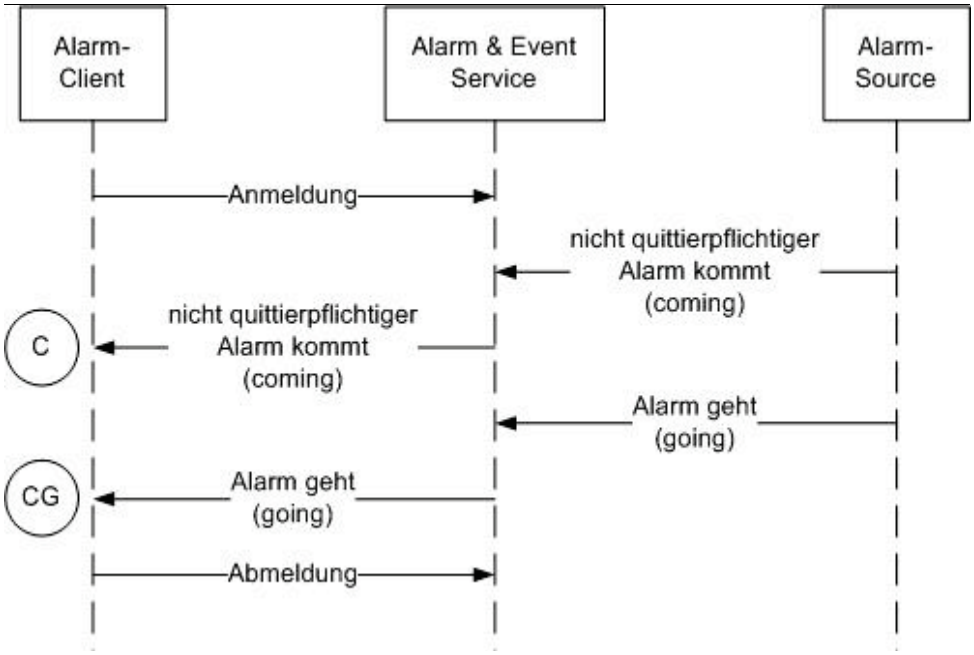


图 6-8:无需应答的报警的信号流

6.2.3 无状态报警

从报警客户端的角度来看，无状态报警(stateless alarm)具有无需应答报警的状态机。但是与后者不同的是，无状态报警的报警源只输出出现事件。
有两种方法可以输出消失事件：
一种方法是从报警客户端输出，这一方法相当于显式删除报警。另一种方法是报警与事件服务本身在超时后删除报警。
无状态报警的存在是有历史原因的，只能用于 OEM 应用。
比如，目前 HMI 高级版的 MMC 报警（ALARMMSG()命令）是通过 cancel 按钮删除的，也就是上文提及的第一种方法。

注

无状态报警永远无需应答。

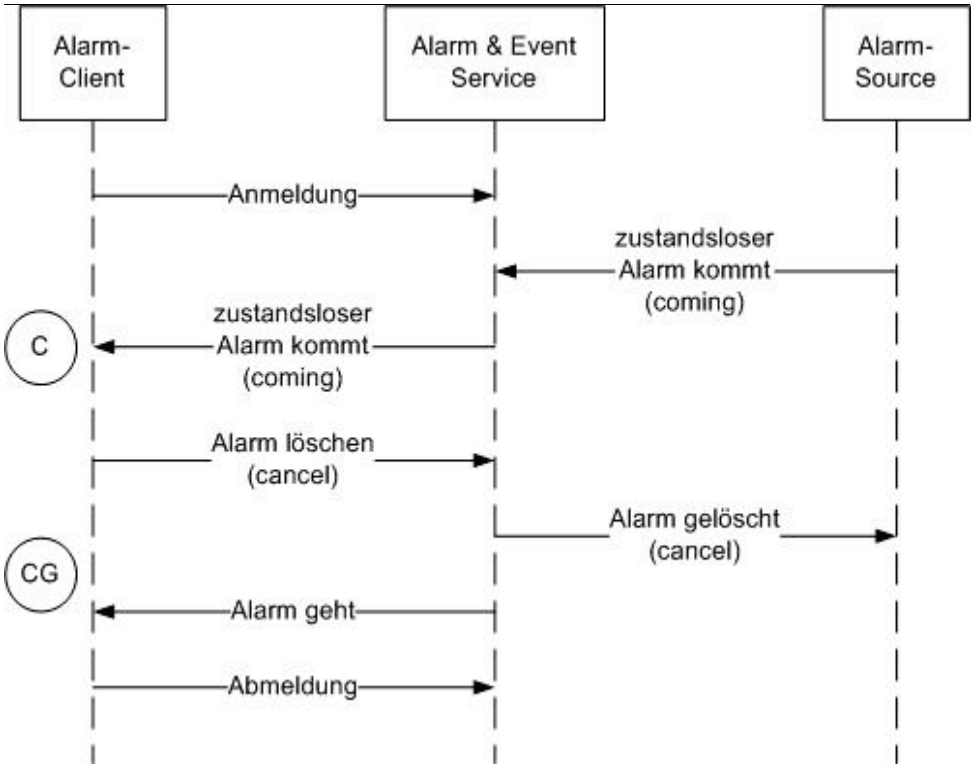


图 6-9:无状态报警由客户端删除时的信号流

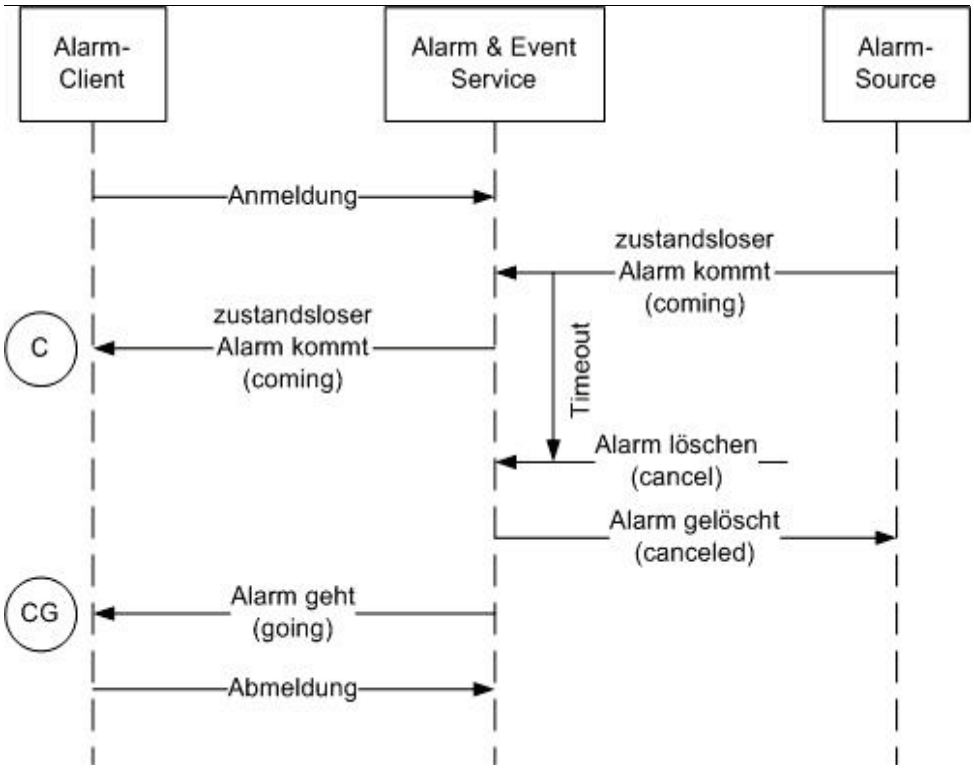


图 6-10:无状态报警由超时删除时的信号流

6.3 分步示例

以下章节将分步介绍报警与事件服务的不同应用区域。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例。

表 6-2：分步示例

应用	示例	章节
访问报警列表。 显示当前待处理的报警。	slexaealarmlist	6.3.2 “显示当前所有待处理的报警（报警列表）”
实现一个 EventSink。 显示所有发生的事件（出现、消失和应答）。	slexaeeventsink	6.3.3“显示当前所有发生的单独事件（EventSink）”
实现报警源(EventSource)。创建 HMI 报警。	slexaeeventsource	6.3.4“创建 HMI 报警 (EventSource)”

编程包中也包含了以下其他可执行的示例程序。这些程序可以由“分步示例”中导出。

表 6-3：其他示例

应用	示例
访问事件列表。 显示所有发生的单独事件。	slexaeeventlist

此处只举例说明或描述与报警与事件服务相关的内容。源文件的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

开发环境

为报警与事件服务示例程序建立开发环境所需的步骤和 5.3 章指出的步骤完全一致。

6.3.1 准备

概述

完成以下指出的准备工作后，才能从自定义的项目或类中调用报警与事件服务。

检查库

在项目设置中检查库 `slaesvcadapter.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `slaesvcadapter.lib`

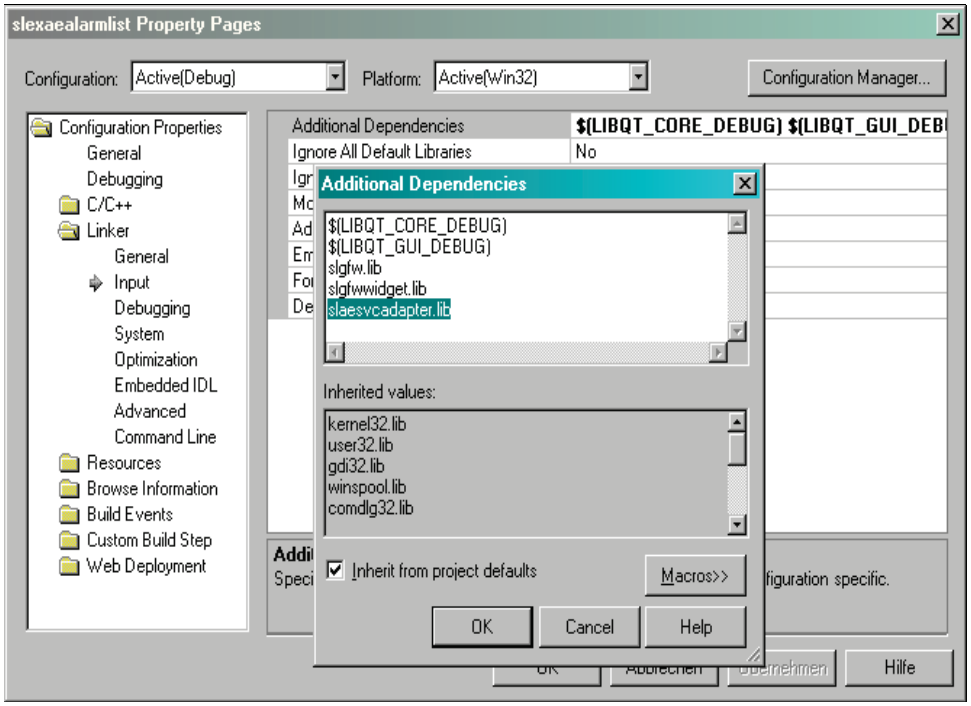


图 6-11:库“slaesvcadapter.lib”

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lslaesvcadapter
```

头文件

要使用报警与事件服务的客户端需要使用一份和实际应用配套的头文件：

表 6-4：所需的头文件

应用场合	头文件
SlAeQAlarmPtrList	#include "slaeqalarmptrlist.h"
SlAeQEventPtrList	#include "slaeqeventptrlist.h"
SlAeQEventSink	#include "slaeqeventsink.h"
SlAeQEventSource	#include "slaeqeventsource.h"

6.3.2 显示当前所有待处理的报警（报警列表）

概述

下面的示例分步展示了如何实现对报警列表的访问，此处以报警特性时间戳、报警号和报警文本为代表进行说明。

注

SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExAeAlarmList”。

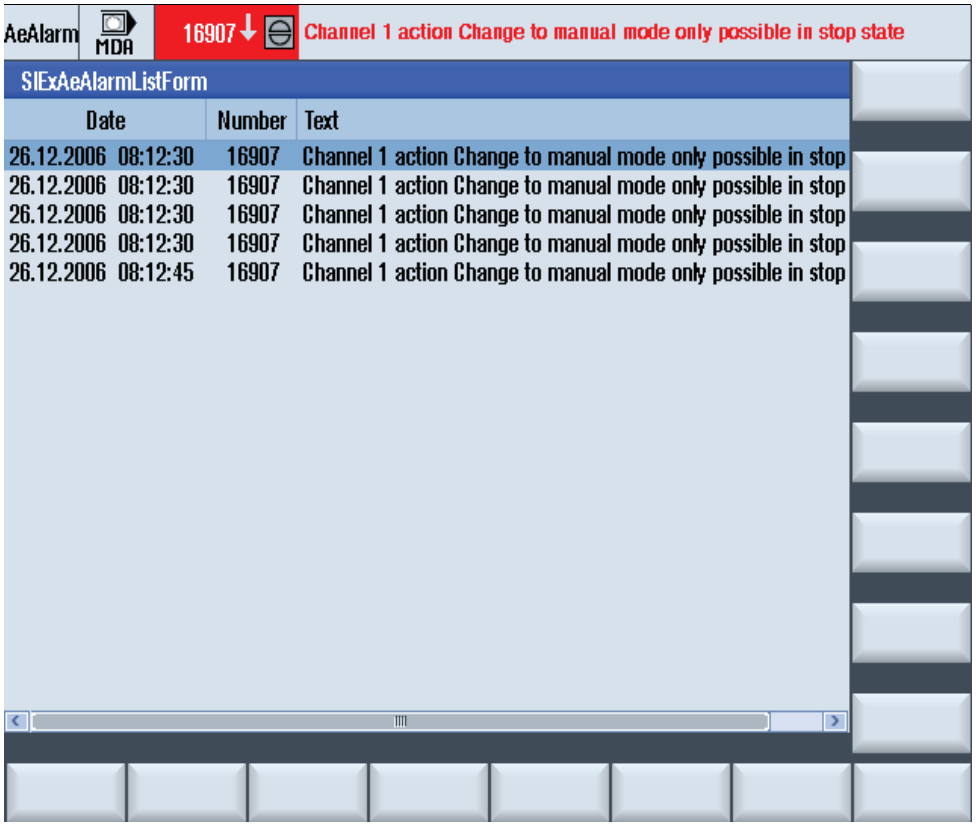


图 6-12:示例 SIExAeAlarmList

务必要满足章节 6.3.1 列出的前提条件。

第 1 步

作为私有成员变量创建指向 SlAeQAlarmPtrList 类中一个对象的指针。

```
SlAeQAlarmPtrList* m_pQAlarmPtrList;
```

第 2 步

成员函数（槽）用作 SlAeQAlarmPtrList 类的回调函数。

```
private slots:
    void onListChanged();
```

第 3 步

在客户端的构造函数中创建一个 SlAeQAlarmPtrList 类的实例。

```
m_pQAlarmPtrList = new SlAeQAlarmPtrList(this);
```

第 4 步

首先要初始化刚刚创建的 SlAeQAlarmPtrList 类的实例，然后才能使用它。

```
long lError = m_pQAlarmPtrList->init();
```

第 5 步

除了标准特性“AlarmID”和“Timestamp”外，本例中还要提供某语言的报警/信息文字。为此要在列表中填入预定义的附加特性（返回特性）。

```
SlAeIdsList attributes2Subscribe;
attributes2Subscribe.push_back(SLAE_EV_ATTR_MSGTEXT);
```

第 6 步

通常情况下，在所有报警类别中将该列表设为返回特性。

```
SlAeCategoryInfoArray rCategories;
rCategories.clear();
long lError = m_pQAlarmPtrList->queryCategories(rCategories);

SlAeCategoryInfoArray::Iterator it;
for(it = rCategories.begin(); it != rCategories.end(); ++it)
{
      lError = m_pQAlarmPtrList->setReturnAttributes((*it).m_nId,
                                                      attributes2Subscribe);
}
```

第 7 步

将成员函数（槽）与报警列表的信号 alarmListChanged 关联在一起，以便接收关于报警列表变化的通知。

```
QObject::connect(m_pQAlarmPtrList, SIGNAL(listChanged()),
   this, SLOT(onListChanged()));
```

第 8 步

现在激活报警列表后，槽 alarmListChanged 会收到关于报警列表变化的通知。

```
m_pQAlarmPtrList->activate();
```

第 9 步

调用立即返回，槽 onAlarmListChanged 收到关于报警列表变化的通知。在一个循环内对报警列表的元素进行迭代，计算出单独的报警。

```
QList<SlAeEvent*>::iterator it;

for(it = m_pQAlarmPtrList->getList()->begin();
    it != m_pQAlarmPtrList->getList()->end();
    ++it)
{
   SlAeEvent* pAEEvent = *it;

   QDateTime dtTimestamp = pAEEvent->getTimestamp();
   quint32 nAlarmId = pAEEvent->getAlarmId();
   QVariant vMsgText = pAEEvent->getAttribute(SLAE_EV_ATTR_MSGTEXT);

pAEEvent = 0;
}
```

第 10 步

封锁报警列表后，报警与事件服务不再向客户端发送任何关于列表变化的通知。

```
m_pQAlarmPtrList->deactivate();
```

第 11 步

不再需要报警列表时，必须关闭并删除该列表。

```
m_pQAlarmPtrList->fini();
delete m_pQAlarmPtrList;
m_pQAlarmPtrList = 0;
```

6.3.3 显示当前所有发生的单独事件（EventSink）

概述

下面的示例分步展示了如何实现对 EventSink 的访问，此处以报警特性时间戳、报警号、报警状态和报警文本为代表进行说明。

注

SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExAeEventSink”。

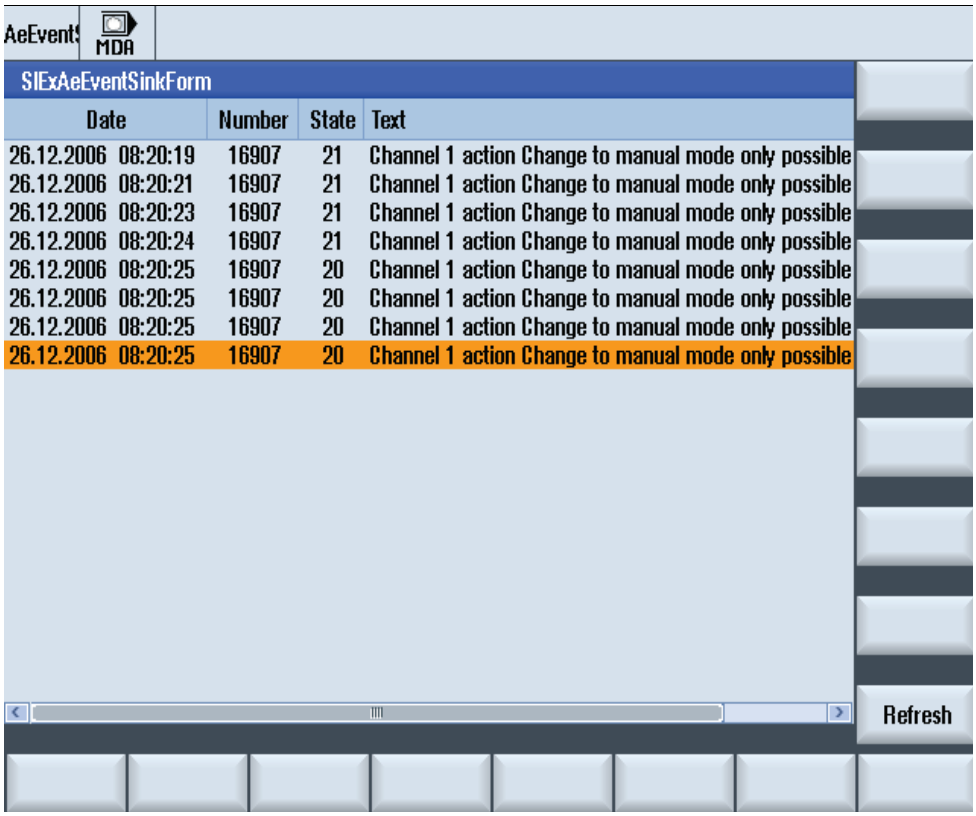


图 6-13:示例 SIAeEventSink

务必要满足章节 6.3.1 列出的前提条件。

第 1 步

作为私有成员变量创建指向 SlAeQEventSink 类中一个对象的指针。

```
SlAeQEventSink* m_pQEventSink;
```

第 2 步

成员函数（槽）用作 SlAeQEventSink 类的回调函数。

```
private slots:
    void onNewEvent(const QList<SlAeEvent*>&);
```

第 3 步

在客户端的构造函数中创建一个 SlAeQEventSink 类的实例。

```
m_pQEventSink = new SlAeQEventSink(this);
```

第 4 步

首先要初始化刚刚创建的 SlAeQEventSink 类的实例，然后才能使用它。

```
long lError = m_pQEventSink->init("eng");
```

第 5 步

除了标准特性“AlarmID”、“Timestamp”和“AlarmState”外，本例中还要提供某语言的报警/信息文字。为此要在列表中填入预定义的附加特性（返回特性）。

```
SlAeIdsList attributes2Subscribe;
attributes2Subscribe.push_back(SLAE_EV_ATTR_MSGTEXT);
```

第 6 步

通常情况下，在所有报警类别中将该列表设为返回特性。

```
SlAeCategoryInfoArray rCategories;
rCategories.clear();
long lError = m pQEventSink->queryCategories(rCategories);

SlAeCategoryInfoArray::Iterator it;
for(it = rCategories.begin(); it != rCategories.end(); ++it)
{
lError = m_pQEventSink->setReturnAttributes((*it).m_nId,
                                                    attributes2Subscribe);
}
```

第 7 步

将成员函数（槽）与报警列表的信号 newEvent 关联在一起，以便接收关于发生的事件的通知。

```
QObject::connect(m_pQEventSink,
                          SIGNAL(newEvent(const QList<SlAeEvent*>&)),
                          this,
                          SLOT(onNewEvent(const QList<SlAeEvent*>&)));
```

第 8 步

现在激活 EventSink 后，槽 onNewEvent 会收到关于发生的事件的通知。

```
m_pQEventSink->activate();
```

第 9 步

调用立即返回，槽 onNewEvent 收到关于发生的事件的通知。将 SlAeEvent 类型的事件 event 的指针作为传递函数传送，然后计算该指针。

```
QDateTime dtTimestamp = event->getTimestamp();
quint32 nAlarmId = event->getAlarmId();
SlAeAlarmStateEnum nAlarmState = event->getAlarmState();
QVariant vMsgText = event->getAttribute(SLAE_EV_ATTR_MSGTEXT);
```

第 10 步

封锁 EventSink 后，暂停关于发生的事件的通知。

```
m_pQEventSink->deactivate();
```

第 11 步

不再需要 EventSink 时，必须关闭并删除该列表。

```
m_pQEventSink->fini();
delete m_pQEventSink;
m_pQEventSink = 0;
```

6.3.4 创建 HMI 报警(EventSource)

下面的示例分步展示了如何实现一个 EventSource（报警源）以输出 HMI 报警。

注

SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SlExAeEventSource”。

务必要满足章节 6.3.1 列出的前提条件。

另外还需要在报警与事件服务器的配置文件中完成以下四项设置：

1. 自定义报警源
2. 添加报警与事件服务器的数据库
3. 自定义报警文本
4. 为报警与事件服务器添加自定义的报警文本

自定义报警源

在和语言无关的数据库中，每条 OEM 报警必须引入一个新的 MSGTEXT 特性，作为某语言的报警文本的引用。siemens/cfg 目录下的数据库文件(slaedatabase.hmi)禁止更改（在升级时可能会被改写），因此要首先以 OEM 自定义名称创建一份数据库文件，只在其中加入新的 MSGTEXT 特性，然后将该文件保存到 oem 目录下。数据库文件始终要以二进制格式保存到该目录下。

利用下述命令将数据库文件从 XML 格式转换到 HMI 格式（二进制格式）：

```
slhmiconvertercmd    -in slaeeventsource_db.xml -out slaeeventsource_db.hmi -
converter slaexmltobinconverter.SlAeBinConverter
```

当然，也可以从二进制格式转换为 XML 格式。

如何自定义数据库见：示例 slexaeeventsource_db.hmi

```
<?xml version="1.0" encoding="UTF-8" ?>
<SlAeAlarmAttributes Version="01.00.00.00">
  <Types>
    <Type TypeName="Condition" TypeID="1">
      <Category Version="1.0" CatID="1">
        <CatDescr>Alarms of example slexaeeventsource.</CatDescr>
      </Category>
    </Type>
  </Types>
  <Sources>
    <Source SourceID="100000" SourceURL="/HMI/SLEXAEEVENTSOURCE" CatLink="1">
      <Alarms DISPLOC="0">
        <HELPFILENAME>sinumerik alarm hmi</HELPFILENAME>
        <Alarm AlarmID="130000">
          <MSGTEXT>slexaeeventsource|Source_130000</MSGTEXT>
        </Alarm>
        <Alarm AlarmID="130001">
          <MSGTEXT>slexaeeventsource|Source 130001</MSGTEXT>
        </Alarm>
      </Alarms>
    </Source>
  </Sources>
</SlAeAlarmAttributes>
```

说明：
上例自定义了一个报警源，其源 URL（SourceURL）为“/HMI/SLEXAEEVENTSOURCE”、源 ID（SourceID）为 100000 而类别 ID（CatLink）为 1。
另外，该报警源会发出一条报警，其报警号（AlarmID）为 130000 和 130001，某语言的报警文本保存在文本文件“slexaeeventsource_alarmtext_<语言代码>.qm”中。

MSGTEXT 特性的映射规定为：

<上下文 ID>|<文本 ID>

上下文 ID 指某语言文件中使用的文本上下文，利用它可以确定语言文件中和语言相关的文本（例如：slexaeeventsource）。
文本 ID 指指向某语言文件中和语言相关的文本的引用。
（另见下文“自定义报警文本”）

添加报警与事件服务器的数据库

在报警与事件服务的设置文件(slaesvcconf.xml/.cfg)中添加条目来声明 OEM 自定义的数据库。此处同样不允许在 siemens 目录下的文件中添加条目，而是应在 oem 目录下创建一份拷贝件，然后只在其中添加声明新建数据库的条目。

在“CONFIGURATION/DataBases”下指定构成报警与事件服务数据库的所有文件。该数据库包含了和语言无关的报警特性定义和特性值。添加更多文件便可以添加 OEM 数据库。

在报警与事件服务的设置文件“slaesvcconf.xml”中添加了数据库后，该文件要以二进制格式保存到目标目录下。利用下述命令将该文件从 XML 格式转换到 CFG 格式（二进制格式）：

```
slhmiconvertercmd         -in slaesvcconf.xml -out slaesvcconf.cfg -converter
slhmisettingsconverter.SlHmiSettingsConverter
```

当然，也可以将该文件从二进制格式转换为 XML 格式。

下面的例子“slexaeeventsource_db.hmi”展示了如何添加自定义数据库：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Configuration of the Solutionline Alarm & Event Service -->
<CONFIGURATION>
  <DataBases>
    <MyDataBase type="QString" value="slexaeeventsource db.hmi"/>
  </DataBases>
</CONFIGURATION>
```

注

数据库基本上可以任意命名， 但“DataBase_01”除外，它预留给西门子使用。

自定义报警文本

和语言相关的报警文本必须作为 TS 格式的 SINUMERIK Operate 文本文件提供。该文本文件通过“slhmiconvertercmd.exe”或“slhmiconvertergui.exe”转换为二进制格式后保存到 oem 的子目录 lng 下。
报警与事件服务借助特性 MSGTEXT 从数据库中找到对应的报警文本。此处必须指定文本上下文 ID（TS/context/nam）和文本 ID（TS/context/message/source），详见上文“自定义报警源”中指出的 MSGTEXT 特性的映射规定。

报警文本文件“slaeeventsource_alarmtext_eng.ts”

```
<!DOCTYPE TS>
<TS>
   <context>
     <name>slexaeeventsource</name>
     <message>
        <source>Source_130000</source>
        <translation>This is alarmtext &quot;130000&quot; of application
                                &quot;slexaeeventsource&quot;.</translation>
     </message>
   </context>
</TS>
```

报警文本文件要以二进制格式保存在目标目录下。利用下述命令将该文件从 TS 格式转换到 QM 格式（二进制格式）：

```
slhmiconvertercmd.exe -in slaeeventsource_alarmtext_eng.ts -out
slaeeventsource_alarmtext_eng.qm -converter sltxtconverter.SlTxtConverter
```

为报警与事件服务器添加自定义的报警文本

在报警与事件服务的设置文件(slaesvcadapconf.xml/.cfg)中添加条目，来声明 OEM 自定义的、和语言相关的 OEM 报警文本文件。此处同样不允许在 siemens 目录下的文件中添加条目，而是应在 oem 目录下创建一份拷贝件，然后只在其中添加自定义的报警文本。

创建自定义报警文本时，必须在报警与事件服务的配置文件(slaesvcadapconf.xml)中添加条目。
在“CONFIGURATION/AlarmTexts/BaseNames”下指定多个数据库的名称，目前只有一个条目“BaseName_01”。这些条目决定了在启动时会加载哪些报警文本文件。因为数据库的名称和报警文本文件的名称是一致的（比如：对于英语的报警文本文件“slexaeeventsource_alarmtext_eng.qm”而言，数据库名称为不带语言代码的名称“slexaeeventsource_alarmtext”。OEM 可以在此添加数据库。但要使用自定义的数据库名称。对应的 OEM 报警文本文件必须位于 oem 或 user 下的 lng 子目录。

更多关于自定义报警文本的信息可参见“SINUMERIK Operate 调试手册”的章节 5.4“设计用户报警文本”。

下面的例子“slexaeeventsource_alarmtext”展示了如何添加自定义的报警文本：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Configuration of the Solutionline Alarm & Event Service Adapter -->
<CONFIGURATION>
  <AlarmTexts>
    <BaseNames>
      <MyDataBase type="QString" value="slexaeeventsource_db.hmi"/>
    </BaseNames>
  </AlarmTexts>
</CONFIGURATION>
```

注

数据库基本上可以任意命名， 但“BaseName_01”除外，它预留给西门子使用。

在报警与事件服务的设置文件“slaesvcadapconf.xml”中添加了报警文本后，该文件要以二进制格式保存到目标目录下。利用下述命令将该文件从 XML 格式转换到 CFG 格式（二进制格式）：

```
slhmiconvertercmd          -in slaesvcadapconf.xml -out slaesvcadapconf.cfg -
converter slhmisettingsconverter.SlHmiSettingsConverter
```

当然，也可以将该文件从二进制格式转换为 XML 格式。

第 1 步

作为私有成员变量创建指向 SlAeQEventSource 类中一个对象的指针。

```
SlAeQEventSource* m_pQEventSource;
```

第 2 步

在客户端的构造函数中创建一个 SlAeQEventSink 类的实例。

```
m_pQEventSource = new SlAeQEventSource();
```

第 3 步

首先要初始化并借着激活刚刚创建的 SlAeQEventSource 类的实例，然后才能使用它。
此处源 ID 必须作为参数传送，通常该 ID 是在 OEM 号段中选择的。更多关于指定源 ID 的信息参见章节 6.5.6“SlAeEvent（单独事件）- 报警源”。

```
long lError = m_pQEventSource->init(100000);
lError      = m_pQEventSource->activate();
```

第 4 步

将成员函数（槽）eventAcknowledged 与 EventSource 的信号 eventAcknowledged 关联在一起，以便接收关于报警应答的通知（针对需应答的报警）。一种应答方式是按下“诊断”区“报警列表”屏幕中的“应答报警”软键。另一种应答方式是调用通用的订阅函数 acknowledgeEvent，另见章节 6.5.4 “通用的订阅函数”。

```
QObject::connect(m pQEventSource,
                     SIGNAL(eventAcknowledged(const SlAeEvent&)),
                     this,
                     SLOT(eventAcknowledged(const SlAeEvent&)));
```

第 5 步

将成员函数（槽）cancelAlarms 与 EventSource 的信号 cancelAlarms 关联在一起，以便接收关于报警删除操作的通知。一种删除方式是按下“Alarm cancel”键（即“BigMac”键），另一种是按下回调键“^”。

```
QObject::connect(m_pQEventSource,
                     SIGNAL(cancelAlarmGroup(const long)),
                     this,
                     SLOT(cancelAlarmGroup(const long)));
```

第 6 步

将成员函数（槽）cancelAlarmGroup 与 EventSource 的信号 cancelAlarmGroup 关联在一起，以便删除某个报警源发出的、需要成组删除（CancelGroup）的报警。

```
QObject::connect(m pQEventSource,
                     SIGNAL(cancelAlarmGroup(const long)),
                     this,
                     SLOT(cancelAlarmGroup(const long)));
```

第 7 步

将成员函数（槽）alarmCanceled 与 EventSource 的信号 alarmCanceled 关联在一起，以便接收关于由报警与事件服务自动完成的删除事件（消失事件）的通知。此类事件具体指状态为 SLAE_ALARM_HMI_CANCEL_TIME_COMING 或 SLAE_ALARM_HMI_CANCEL_BTN_COMING 的两种 HMI 报警。

```
QObject::connect(m_pQEventSource,
                     SIGNAL(alarmCanceled(const long, const SlAeEvent&)),
                     this,
                     SLOT(alarmCanceled(const long, const SlAeEvent&)));
```

第 8 步

原则上一个 HMI 报警源可以输出 4 种报警，但具体报警类型要根据实际应用配置：

- **Stateless alarm**（无状态报警）
  在下述两个无状态报警示例中，报警源只输出“出现事件”。与出现事件对应的“消失事件”由报警与事件服务自动输出，即：报警源无需负责报警的删除。然而报警源可借助报警与事件服务发出的信号 `alarmCanceled` 得知报警已被删除这一事件。
  但在理论上无状态报警可以提前删除，一种方法是按下“诊断”区“报警列表”屏幕中的“删除 HMI 报警”软键。另一种方法是调用通用的订阅函数 `cancelAlarm`，另见章节 6.5.4 “通用的订阅函数”。
  - **“HMI Cancel Button”类报警**指通过按下“Alarm cancel”键（即“BigMac”键）删除的报警。

    出现事件

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130000);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_HMI_CANCEL_BTN_COMING);
aeEvent.setCategory(SLAE_EV_CAT_SINUMERIK_HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

    消失事件

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130000);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_HMI_CANCEL_BTN_CAME_GOING);
aeEvent.setCategory(SLAE_EV_CAT_SINUMERIK_HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

  - **“HMI Cancel Time”类报警**指在报警与事件服务中规定的时间期满后自动删除的报警。

    出现事件

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130001);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_HMI_CANCEL_TIME_COMING);
aeEvent.setCategory(SLAE EV CAT SINUMERIK HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

    消失事件

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130001);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE ALARM HMI CANCEL TIME CAME GOING);
aeEvent.setCategory(SLAE EV CAT SINUMERIK HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

- **无需应答的报警**
  此类报警可按下“诊断”区“报警列表”屏幕中的“删除 HMI 报警”软键删除。另一种删除方法是调用通用的订阅函数 cancelAlarm，另见章节 6.5.4 “通用的订阅函数”。
  在这两种方法中，输出对应“消失事件”的报警源都会调用信号 cancelAlarmGroup。

  出现事件

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130002);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_COMING);
aeEvent.setCategory(SLAE_EV_CAT_SINUMERIK_HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

  消失事件

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130002);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_CAME_GOING);
aeEvent.setCategory(SLAE_EV_CAT_SINUMERIK_HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

- **需应答的报警**
  此类报警可按下“诊断”区“报警列表”屏幕中的“删除 HMI 报警”软键删除。另一种删除方法是调用通用的订阅函数 cancelAlarm，另见章节 6.5.4 “通用的订阅函数”。
  在这两种方法中，输出对应“消失事件”的报警源都会调用信号 cancelAlarmGroup。
  正如它的名字表明的一样，此类报警是需要强制应答的。一种应答方式是按下“诊断”区“报警列表”屏幕中的“应答报警”软键。另一种应答方式是调用通用的订阅函数 acknowledgeEvent，另见章节 6.5.4 “通用的订阅函数”。
  在这两种方法中，报警源都会调用信号 eventAcknowledged。

! **重要提示**

同一个报警源输出的、需应答的 HMI 报警不允许有多个实例。这是因为在某些条件下报警与事件服务无法明确 HMI 报警的出现事件的含义。
示例：
一条需应答的 HMI 报警已消失，但未经应答。如果此时报警源要输出下一个出现事件，报警与事件服务就无法明确地判定这是第一个报警实例的出现（二次出现）事件还是新实例的出现事件。

出现事件：

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130003);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE ALARM COMING TOACK);
aeEvent.setCategory(SLAE EV CAT SINUMERIK HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

消失事件

根据报警的当前状态输出与出现事件对应的消失事件。

- 当前状态：SLAE_ALARM_COMING_TOACK

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130003);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_CAME_GOING_TOACK);
aeEvent.setCategory(SLAE_EV_CAT_SINUMERIK_HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

- 当前状态：SLAE_ALARM_CAME_ACKING

```
SlAeEvent aeEvent;

aeEvent.setAlarmId(130003);
aeEvent.setSourceId(100000);
aeEvent.setAlarmState(SLAE_ALARM_CAME_ACKED_GOING);
aeEvent.setCategory(SLAE_EV_CAT_SINUMERIK_HMI);

long lError = m_pQEventSource->createEvent(aeEvent);
```

第 10 步

报警与事件服务会自动在报警源中生成一张列表，列明所有该源输出的报警，以备使用。
编程人员因此无需记住报警源当前输出了哪些报警。
尤其是在需要成组删除信号 `cancelAlarmGroup` 中确定的报警时，该列表可快速找出这些报警。

! **重要提示**

此处我们推荐使用 Java 风格的迭代器 `QListIterator`，因为在某些情况下进行迭代时，列表可能会因诸如消失事件的输出而变化。
使用 STL 风格的迭代器肯定会导致程序异常终止，因为一旦列表变化迭代器会立即变为无效。详细信息参见 Qt 的文档。

```
QList<SlAeEvent*>*          pList = m pQEventSource->getList();
QListIterator<SlAeEvent*>   it(*pList);

it.toFront();

while(it.hasNext())
{
    SlAeEvent*  pAEEvent = it.next();
    SlAeEvent*  pAEEventClone = 0;

    QVariant vCancelGroup ;
    vCancelGroup = pAEEvent->getAttribute(SLAE EV ATTR CANCELGROUP);

    if ( lCancelGroup == vCancelGroup.toLongLong() )
    {
        //do something
    }
}
```

第 11 步

不再需要 EventSource 时，必须关闭并删除该列表。

```
m pQEventSource->deactivate();
m_pQEventSource->fini();
delete m_pQEventSource;
m_pQEventSource = 0;
```

6.4 集成自定义报警（HMI 报警和 PLC 报警）的在线帮助

一览

需要为自定义的 HMI 报警或 HMI 报警设计在线帮助时，您可以采用帮助服务的机制。设计了在线帮助后，在标准操作区“诊断”内按下操作面板（TCU 或 OP）上的 INFO 键（帮助键），便可显示一份和语言相关的 HTML 格式的在线帮助文件。

设计示例位于 AE-Service\SIExAeEventSource 的示例目录下。该示例为 HMI 报警设计了在线帮助。比如：在输出 HMI 报警 130000 后，便可在“诊断”操作区内按下 INFO 键来查看为该报警设计的帮助文本。

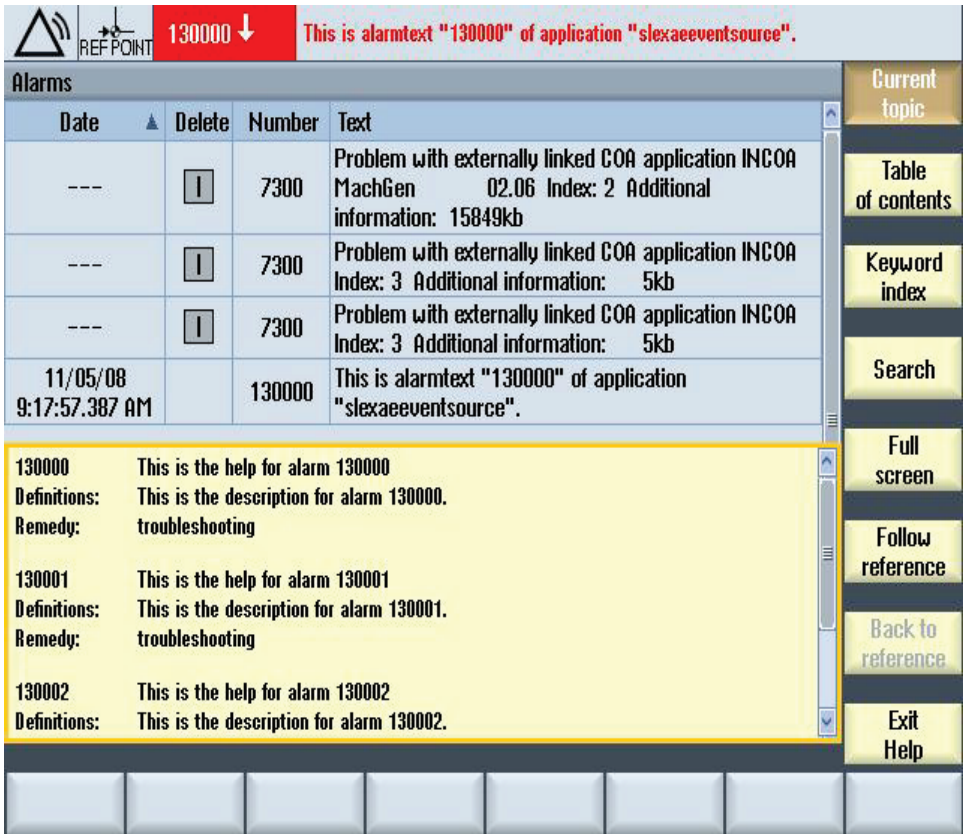


图 6-14:设计了在线帮助的标准操作区“诊断”

要实现这种在线帮助您需要完成以下两个步骤。

创建帮助文件

帮助文件要以 HTML 格式和支持的各种语言创建。HTML 文件的缺省名称为：

自定义的 PLC 报警➔ “sinumerik_alarm_oem_plc_pmc.html”
自定义的 HMI 报警➔ “sinumerik_alarm_oem_hmi.html”

在 HTML 文件中作为 HMTL 锚点指定报警号，锚点是显示帮助时直接跳转到的位置。

目录“hlp\<lng>\sinumerik_alarm_hmi\sinumerik_alarm_oem_hmi.html”下有一个示例 SIExAeEventSource，展示了如何创建这样一份帮助文件。

创建帮助手册（仅限帮助内容分布在多个 HMTL 文件中的情况）

希望在线帮助分布在多个 HTML 文件中时，还需要创建一份帮助手册。如果自定义的帮助手册仅在一份 HTML 文件中创建，便可跳过该步骤！

在帮助手册中确定哪个 HTML 文件对应哪个报警号段。帮助手册的缺省名称为：

用于自定义 PLC 报警：➔ “sinumerik_alarm_plc_pmc.xml”

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE HMI SL HELP>
<HMI SL HELP>
  <BOOK>
    <NUM_AREAS>
      <NUM_AREA from="130000" to="130999" ref="sinumerik_oem_1.html" />
      <NUM_AREA from="131000" to="135999" ref="sinumerik_oem_2.html" />
      <NUM_AREA from="136000" to="139999" ref="sinumerik_oem_3.html" />
    </NUM_AREAS>
  </BOOK>
</HMI_SL_HELP>
```

用于自定义 HMI 报警：➔ “sinumerik_alarm_hmi.xml”

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE HMI SL HELP>
<HMI_SL_HELP>
  <BOOK>
    <NUM_AREAS>
      <NUM_AREA from="500000" to="599999" ref="sinumerik_oem_1.html" />
      <NUM_AREA from="600000" to="699999" ref="sinumerik_oem_2.html" />
      <NUM_AREA from="700000" to="799999" ref="sinumerik_oem_3.html" />
      <NUM_AREA from="800000" to="899999" ref="sinumerik_oem_4.html" />
    </NUM_AREAS>
  </BOOK>
</HMI_SL_HELP>
```

复制创建的文件

现在您可以将创建好的文件复制到系统中：

1. 在路径“<安装路径>/oem/sinumerik/hmi/hlp”下创建一个目录。目录的名称要和语言代码一致。

   比如说，要集成德语版和英语版的帮助文件时，应创建以下目录：
   “<安装路径>/oem/sinumerik/hmi/hlp/deu”
   “<安装路径>/oem/sinumerik/hmi/hlp/eng”。

2. 在该目录中创建以下子目录：

   用于自定义的 PLC 报警➔“sinumerik_alarm_plc_pmc”
   用于自定义的 HMI 报警➔“sinumerik_alarm_hmi”

3. 接着将对应的 HTML 文件复制到子目录中，将可能有的帮助手册直接复制到对应语言的子目录中。

不含帮助手册的目录结构：

假设自定义了 PLC 报警和 HMI 报警，那么德语版本帮助手册的目录结构应为：

图 6-15: 不含帮助手册的德语目录结构

含帮助手册的目录结构：

假设只是自定义了 HMI 报警，那么德语版本帮助手册的目录结构应为：

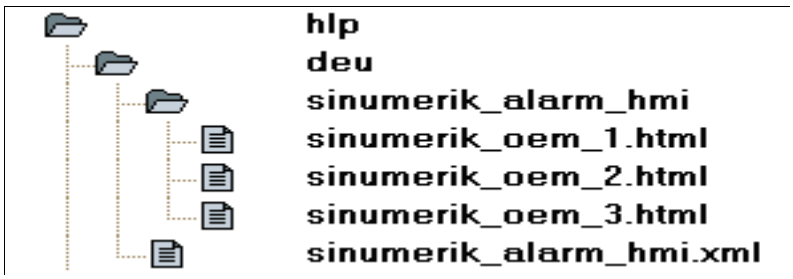


图 6-16: 含帮助手册的德语目录结构

重启 SINUMERIK Operate 后，您便可以在标准操作区“诊断”中调用在线帮助。

注

为加快处理，目录**“<安装路径>/siemens/sinumerik/sys_cache/hmi/hlp”**下会生成一份二进制格式的帮助文件。 一旦在线帮助有所改动，必须删除帮助文件及其模板“slhlp_sinumerik_alarm_plc_pmc_*.hmi”或“slhlp_sinumerik_alarm_hmi_*.hmi”。

备选方法

您也可以在一份 HTML 文件中创建针对一个报警号的在线帮。 文件名称因此只包含了报警号，比如 130300.html 代表了报警 130300 的帮助文件。

将该 HTML 文件保存到上文指出的目录下：

针对自定义的 PLC 报警➔“hlp\<语言代码>\sinumerik_alarm_plc_pmc”
针对自定义的 HMI 报警➔“hlp\<语言代码>\sinumerik_alarm_hmi”

6.5 接口和对象

下图展示了报警与事件服务提供的所有类和下属方法。

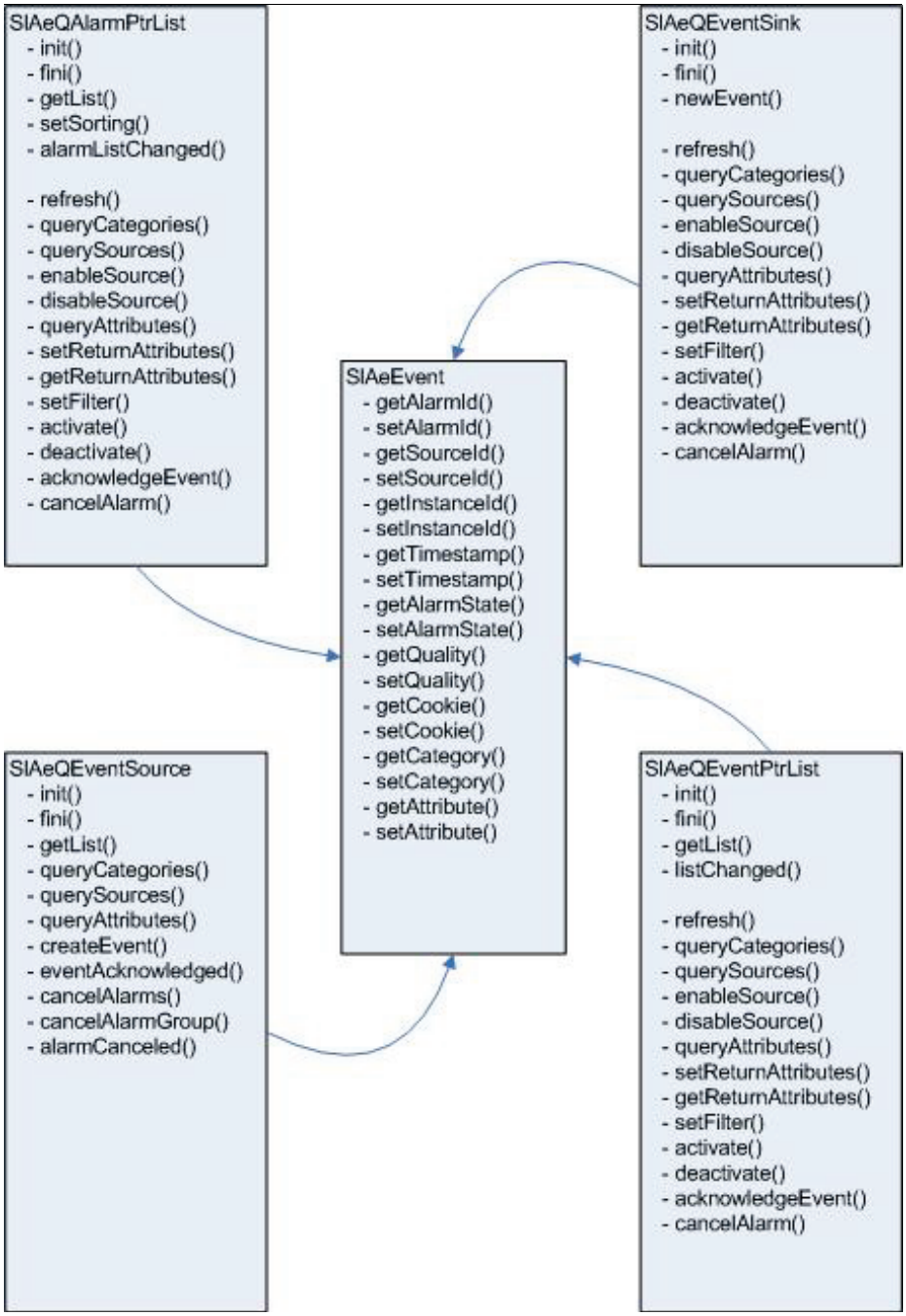


图 6-17:报警与事件服务提供的类与函数一览

6.5.1 报警列表的订阅(SlAeQAlarmPtrList)

该类向客户端返回一张当前所有待处理报警的列表，其中指出了报警的状态、报警的所有特性以及某语言的报警文本。

构造函数

只存在默认构造函数。无法复制该对象或对其赋值。

初始化

在创建了该类的一个实例对象后，必须始终首先调用函数 `init` 来初始化该对象。

表 6-5：初始化

long SlAeQAlarmPtrList::init(void)	
参数	含义
-	-
返回值	错误代码

摧毁对象

最后在摧毁实例对象前必须调用函数 `fini`。

表 6-6：摧毁对象

long SlAeQAlarmPtrList::fini(void)	
参数	含义
-	-
返回值	错误代码

迭代器函数

访问列表单元需要首先调用函数 `getList`。由此可获得一个指向模板类 `QList` 的 const 指针，模板类的类型为 `SlAeQEvent*:QList<SlAeEvent*>`。该模板类会提供迭代器函数，此处的说明引用 Qt 文档的内容。

表 6-7：关于报警列表变化的通知

QList<SlAeEvent*>* const SlAeQAlarmPtrList::getList(void)	
参数 -	含义 -
返回值	报警列表 模板类 QList，类型为 SlAeEvent*

排序

利用函数 `setSorting` 可确定报警的排序方式。报警可以按下述特性升序排列或降序排列：
- 优先级(Severity)
- 时间戳(Timestamp)
- 进入报警与事件服务的的顺序

表 6-8：排序

long SIAeQAlarmPtrList::setSorting(SIAeSvcSorting eSorting, SIAeSvcOrdering eOrdering = SLAE_ORDER_DESCENDING	
参数	含义
eSorting	排序依据： 比如有：优先级、时间戳和进入顺序等 另见章节 6.5.7“枚举数和定义”下的段落“排序”
eOrdering	排序顺序： 升序或降序 另见章节 6.5.7“枚举数和定义”下的段落“排序” 缺省值：降序
返回值	错误代码

关于报警列表变化的通知

利用 Qt 信号 `alarmListChanged` 客户端可获得关于报警列表变化的通知。

表 6-9：关于报警列表变化的通知

void SIAeQAlarmPtrList::alarmListChanged(void)	
参数	含义
-	-
返回值	-

通用的订阅函数

这些函数在章节 6.5.4“通用的订阅函数”中说明。

6.5.2 事件列表的订阅（SIAeQEventPtrList）

该类向客户端返回一张事件的列表，即报警的状态变化列表。每次状态变化（出现、消失、应答）都会向列表加入一条新条目。但列表有容量限制，因此根据 FIFO 原理（先进先出），新事件会删除最旧的条目。

构造函数

只存在默认构造函数。无法复制该对象或对其赋值。

初始化

在创建了该类的一个实例对象后，必须始终首先调用函数 `init` 来初始化该对象。

表 6-10：初始化

long SIAeQEventPtrList::init(quint32 ulMaxSize = 10)	
参数	含义
ulMaxSize	列表单元的最大数目。 缺省值： 10
返回值	错误代码

摧毁对象

最后在摧毁实例对象前必须调用函数 `fini`。

表 6-11：摧毁对象

long SlAeQEventPtrList::fini(void)	
参数	含义
-	-
返回值	错误代码

迭代器函数

访问列表单元需要首先调用函数 `getList`。由此可获得一个指向模板类 `QList` 的 const 指针，模板类的类型为 `SlAeQEvent*:QList<SlAeEvent*>`。该模板类会提供迭代器函数，此处的说明引用 Qt 文档的内容。

表 6-12：

QList<SlAeEvent*>* const SlAeQAlarmPtrList::getList(void)	
参数	含义
-	-
返回值	报警列表 模板类 QList，类型为 SlAeEvent*

关于事件列表变化的通知

利用 Qt 信号 `listChanged` 客户端可获得关于事件列表变化的通知。

表 6-13：关于事件列表变化的通知

void SlAeQAlarmPtrList::listChanged(SlAeEvent* pNewEvent	
参数	含义
pNewEvent	新事件对象的引用
返回值	-

通用的订阅函数

这些函数在章节 6.5.4“通用的订阅函数”中说明。

6.5.3 单独事件的订阅(SlAeQEventSink)

每次报警状态有所变化时（即出现事件、消失事件和应答事件）时，该类利用一个 Qt 信号通知客户端，它不管理列表。客户端收到该类激活后的所有事件。如需收到在此之前发生的所有报警的事件，则需要使用函数 refresh 更新。

! **重要提示**

SlAeQEventSink 对象只能应用在 Qt 主线程中。

构造函数

只存在默认构造函数。无法复制该对象或对其赋值。

初始化

在创建了该类的一个实例对象后，必须始终首先调用函数 `init` 来初始化该对象。

表 6-14：初始化

long SlAeQEventSink::init(const QString& rszLanguage = NULL)	
参数	含义
rszLanguage	指定报警文本的语言，见函数“setReturnAttributes”。没有指定语言或缺少该语言时，此处返回缺省语言的报警文本。HMI 上的语言切换对该订阅没有影响。
返回值	错误代码

摧毁对象

最后在摧毁实例对象前必须调用函数 `fini`。

表 6-15：摧毁对象

long SlAeQEventSink::fini(void)	
参数	含义
-	-
返回值	错误代码

关于新事件的通知

利用 Qt 信号 `newEvent` 客户端可获得关于新事件的通知。

表 6-16：事件列表中的改变

void SlAeQEventSink::newEvent(const QList<SlAeEvent*>& rEventList)	
参数	含义
rEventList	新事件对象的引用
返回值	-

通用的订阅函数

这些函数在章节 6.5.4“通用的订阅函数”中说明。

6.5.4 通用的订阅函数

更新订阅

函数 `refresh` 使订阅发送每个待处理的报警的最新状态（事件）。
该函数可以使订阅进行更新，即向每个客户端通知每个待处理报警的最新状态（事件）。

表 6-17：更新订阅

long refresh(void)	
参数	**含义**
-	-
返回值	错误代码

查询报警类别

该函数可提供关于报警与事件服务中提供的报警类别的信息。属于一个类别的所有报警具有相同的特性组。客户端收到所有报警类别的 ID 和简称。

表 6-18：查询报警类别

long queryCategories(SlAeCategoryInfoArray& rCategories)	
参数	**含义**

rCategories	输出参数：
	SlAeCategoryInfo 结构的 QValueVector 的引用：
	struct SlAeCategoryInfo
	{
	long m_nId; // 类别 ID
	QString m_strDescription; // 说明
	};
	类别 ID 用于查询需要由报警与事件服务提供的报警特性(queryAttributes)或选择报警特性(setReturnAttributes)。
	另见章节 6.5.7“枚举数和定义”下的段落“类别”

返回值	错误代码

查询报警源

该函数可提供关于报警与事件服务中提供的报警源的信息。客户端收到所有报警源的 ID、全名和简称。

注

函数 `setFilters` 用于根据报警源筛选事件。

表 6-19：查询报警源

long querySources(SlAeSourceInfoArray& rSources)	
参数	含义

rSources	输出参数：
	SlAeSourceInfo 结构的 QValueVector 的引用：
	struct SL_AE_SVC_ADAP_EXPORTS SlAeSourceInfo
	{
	long m_nSourceId; // 源 ID
	QString m_strURL; // 源名称
	long m_nCategoryId; // 类别 ID
	};
	另见章节 6.5.7“枚举数和定义”下的段落“源 ID”

返回值	错误代码

激活报警源

该函数用于激活报警源或此前封锁的报警。

! **重要提示**

该函数会影响报警与事件服务的所有订阅。

表 6-20：激活报警源

long enableSource(Q_UINT32 ulSourceId)	
参数	含义
ulSourceId	需要激活的报警源的 ID。 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
返回值	错误代码

封锁报警源

该函数用于封锁整个报警源或单个报警，比如：封锁游荡的报警。

! **重要提示**

在封锁状态下该报警源的所有事件都会丢失，也就是说，即使重新上电或者调用函数 refresh 也无法重新恢复这些事件。
和函数 `setFilter` 相反，该函数会影响报警与事件服务的所有订阅！

注

在报警与事件服务启动后，所有报警源都处于激活状态。

表 6-21：封锁报警源

long disableSource(Q_UINT32 ulSourceId)	
参数	含义
ulSourceId	需要封锁的报警源的 ID。 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
返回值	错误代码

查询特性

该函数可提供关于类别包含的特性的信息。客户端可收到每个特性的 ID、名称、数据类型和简称。

表 6-22：查询特性

long queryAttributes(Q_UINT32 ulCategoryId, SlAeAttributeInfoArray& rAttributes)	
参数	含义
ulCategoryId	需要查询其特性的类别的 ID。

rAttributes	输出参数：
	SlAeAttributeInfo 结构的 QValueVector 的引用：
	struct SL_AE_SVC_ADAP_EXPORTS SlAeAttributeInfo
	{
	long m_nId; // 特性 ID
	QString m_strName; // 特性名称
	QString m_strDescription; // 说明
	};

返回值	错误代码

设置返回特性

该函数用于选择在每次发生一个事件或报警时返回的某一类别中的特性。

重要提示

只有在封锁了订阅后，才允许设置特性。

表 6-23：设置返回特性

long setReturnAttributes(Q_UINT32 ulCategoryId, SlAeIdsList& rAttributeIds)	
参数	含义
ulCategoryId	需要选择其特性的类别的 ID。
rAttributeIds	Long 型 QValueList 的引用。在该列表中要列明特性 ID，在激活订阅后发生每个事件时都会提供这些特性。利用函数 queryAttribute 可查询服务提供的特性。
返回值	错误代码

查询返回特性

该函数提供关于某个类别中选中的特性的信息。

重要提示

无论订阅是在激活状态还是在封锁状态，都可以查询设置的报警特性。

表 6-24：查询返回特性

long getReturnAttributes(Q_UINT32 ulCategoryId, SlAeIdsList& rAttributeIds)	
参数	**含义**
ulCategoryId	需要查询设置的特性的类别的 ID。
rAttributeIds	输出参数： Long 型 QValueList 的引用。在该列表中，函数后列明了设置的报警特性。
返回值	错误代码

设置筛选并删除筛选

该函数用于限制由订阅提供的事件的数量。和函数 `disableSource` 相比，该函数只会影响对应的订阅对象。筛选条件有：优先级(Severity)、报警类别、报警源和事件 ID。每个特性本身也可以用作筛选条件。

表 6-25：设置筛选

long setFilter(Q_UINT32 ulLowSeverity, Q_UINT32 ulHighSeverity, SlAeIdsList& rarrCategories, SlAeIdsList& rarrSources, SlAeIdsList& rarrEventIds)	
参数	**含义**
ulLowSeverity	最低优先级：只发出至少具有该优先级或该优先级以上的报警。如果该值为-1，相当于不设置优先级下限。
ulHighSeverity	最高优先级：只发出具有该优先级或该优先级以下的报警。如果该值为-1，相当不设置优先级上限。
rarrCategories	Long 型 QValueList 的引用，具有允许的类别 ID：只输出具有该类别 ID 的报警。此处指定空列表时，不依据类别筛选报警。 另见章节 6.5.7“枚举数和定义”下的段落“类别”
rarrSources	Long 型 QValueList 的引用，具有允许的源 ID：只输出该源发出的报警。此处指定空列表时，不依据源 ID 筛选报警。
rarrEventIds	Long 型 QValueList 的引用，含允许的事件 ID：只输出含该事件 ID 的报警。此处指定空引用时，不依据事件 ID 筛选报警。
返回值	错误代码

此外，报警或信息也可以是筛选条件。只有一个事件满足两个筛选条件时，才输出该事件，即两个条件进行逻辑与运算。

表 6-26：设置报警和信息的筛选

long setFilter(SlAeFilterTypeEnum eType)	
参数	含义
eType	确定筛选方式。 另见章节 6.5.7“枚举数和定义”下的段落“筛选”
返回值	错误代码

筛选的删除同样也通过函数 `setFilter` 进行：

```
SlAeIdsList catList, sourceList, eventList;
setFilter(-1, -1, catList, sourceList, eventList);
setFilter(SLAE_FILTER_NO_FILTER);
```

激活订阅

函数 `activate` 用于激活订阅。只有激活订阅后，订阅对象才会发出 Qt 信号，报告进入报警与事件服务中新报警的事件或当前待处理报警的状态变化。订阅对象也会报告在激活订阅前仍待处理的报警的事件，以便显示所有当前待处理的报警。

! **重要提示**

订阅激活后无法设置返回特性或筛选。

表 6-27：激活订阅

long activate(void)	
参数	含义
-	-
返回值	错误代码

封锁订阅

该函数封锁订阅。之后可以再次设置返回特性或筛选。

表 6-28：封锁订阅

参数	含义
-	-
返回值	错误代码

应答报警

该函数可使客户端应答需要应答的 SQ 报警或 HMI 报警。

表 6-29：应答由 SlAeEvent 对象的引用确定的事件

long acknowledgeEvent(const SlAeEvent& rEvent)	
参数	含义
rEvent	事件对象的引用。
返回值	错误代码

表 6-30：应答由某个 Cookie 唯一的报警源发出的事件

long acknowledgeEvent(quint32 ulSourceId, quint32 ulCookie	
参数	含义
ulSourceId	发出需要应答的报警的源 ID。 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
ulCookie	需要应答的报警的 Cookie。
返回值	错误代码

表 6-31：应答由唯一的源 ID、报警 ID 与实例 ID 组合确定的事件

long acknowledgeEvent(quint32 ulSourceId, quint32 ulCookie)	
参数	含义
ulSourceId	发出需要应答的报警的源 ID。 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
ulAlarmId	需要应答的报警的 ID（报警号）。
ulInstanceId	需要应答的报警的实例 ID。
返回值	错误代码

删除报警

一个报警源发出（即具有相同源 ID）且属于一个 CancelGroup 组的报警可以成组删除。删除一个报警会自动删除一个报警源发出的、该组内的所有报警。该删除操作是由报警与事件服务执行的。只有报警是无状态报警时，CancelGroup 才起作用。

表 6-32：通过 CancelGroup 删除一个报警

long cancelAlarm(quint32 usSourceId, quint32 ulCancelGroup)	
参数	含义
ulSourceId	发出需要应答的报警的源 ID。 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
ulCancelGroup	需要成组删除的报警(CancelGroup)。
返回值	错误代码

表 6-33：删除报警

long cancelAlarms(quint32 lFlags	
参数	含义
lFlags	删除报警的方式。 另见章节 6.5.7“枚举数和定义”下的段落“删除报警的方式”
返回值	错误代码

6.5.5 报警源(SIAeQEventSource)

SIAeQEventSource 类是应用在 Qt 主线程中的，可使客户端应用程序发出 HMI 报警，模拟 NCK 或 PLC 报警。模拟 NCK 或 PLC 报警与真实报警分开管理，因此无法通过模拟使一个真实的 NCK 报警消失。
通过 SIAeQEventSource 创建的报警也可以是“需要应答的报警”。应答报警后该类会发出信号向报警源通知用户的应答操作。通知采用的是 Qt 的信号与槽机制。信号为 `eventAcknowledged`，始终异步发出。
该类的其他所有函数都以同步方式调用，在调用期间会阻塞 Qt 主线程的执行。

! 重要提示

SIAeQEventSource 对象只能应用在 Qt 主线程中。

构造函数

只存在默认构造函数。无法复制该对象或对其赋值。

初始化

在创建了该类的一个实例对象后，必须始终首先调用函数 `init` 来初始化该对象。

表 6-34：初始化

long SIAeQEventSource::init(qint32 lSourceId = SLAE_HMI_SOURCE_ID)	
参数	含义
lSourceId	指定分给该实例对象的报警源 ID。 100000 以上的号段是预留给 OEM 使用的。 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
返回值	错误代码

接着必须激活 SIAeQEventSource。

表 6-35：激活

long SIAeQEventSource::activate(void);	
参数	含义
返回值	错误代码

摧毁对象

最后在摧毁实例对象前必须调用函数 `fini`。之前还要封锁 SIAeQEventSource。

表 6-36：禁用

long SIAeQEventSource::deactivate(void);	
参数	含义
返回值	错误代码

表 6-37：摧毁对象

long SIAeQEventSource::fini(void)	
参数	含义
-	-
返回值	错误代码

迭代器函数

访问该报警源发出的报警的列表单元需要首先调用函数 `getList`。由此可获得一个指向模板类 `QList` 的 const 指针，模板类的类型为 `SlAeQEvent*:QList<SlAeEvent*>`。该模板类会提供迭代器函数，此处的说明引用 Qt 文档的内容。

! **重要提示**

此处我们推荐使用 Java 风格的迭代器 `QListIterator`，因为在某些情况下进行迭代时，列表可能会因诸如消失事件的输出而变化。
使用 STL 风格的迭代器肯定会导致程序异常终止，因为一旦列表变化迭代器会立即变为无效。

```
SlAeQEventSource*   m pQEvntSrc;

m_pQEvntSrc = new SlAeQEventSource();
m_pQEvntSrc->init(100000)

QList<SlAeEvent*>*         pList = m pQEvntSrc->getList();
QListIterator<SlAeEvent*>   it(*pList);

it.toFront();

while(it.hasNext())
{
            SlAeEvent*  pAEEvent = it.next();

            //do something
}

m pQEvntSrc->fini();
delete m_pQEvntSrc;
m_pQEvntSrc = 0;
```

表 6-38：关于报警列表变化的通知

QList<SlAeEvent*>* const SlAeQAlarmPtrList::getList(void)	
参数	**含义**
-	-
返回值	报警列表 模板类 QList，类型为 SlAeEvent*

查询报警类别

该函数可提供关于报警与事件服务中提供的报警类别的信息。属于一个类别的所有报警具有相同的特性组。客户端收到所有报警类别的 ID 和简称。

表 6-39：查询报警类别

long SlAeQEventSource::queryCategories(SlAeCategoryInfoArray& rCategories)	
参数	含义

rCategories	输出参数：
	SlAeCategoryInfo 结构的 QValueVector 的引用：
	struct SlAeCategoryInfo
	{
	long m_nId; // 类别 ID
	QString m_strDescription; // 说明
	};
	类别 ID 用于查询需要由报警与事件服务提供的报警特性(queryAttributes)或选择报警特性(setReturnAttributes)。
	另见章节 6.5.7“枚举数和定义”下的段落“类别”

返回值	错误代码

查询报警源

该函数可提供关于报警与事件服务中提供的报警源的信息。客户端所有所有报警源的 ID、名称和类别 ID。

表 6-40：查询报警源

long SlAeQEventSource::querySources(SlAeSourceInfoArray& rSources)	
参数	含义

rSources	输出参数：
	SlAeSourceInfo 结构的 QValueVector 的引用：
	struct SL_AE_SVC_ADAP_EXPORTS SlAeSourceInfo
	{
	long m_nSourceId; // 源 ID
	QString m_strURL; // 源名称
	long m_nCategoryId; // 类别 ID
	};
	另见章节 6.5.7“枚举数和定义”下的段落“源 ID”

返回值	错误代码

查询特性

该函数可提供关于类别包含的特性的信息。客户端可收到每个特性的 ID、名称、数据类型和简称。

表 6-41：查询特性

long SlAeQEventSource::queryAttributes(Q_UINT32 ulCategoryId, SlAeAttributeInfoArray& rAttributes)	
参数	含义
ulCategoryId	需要查询其特性的类别的 ID。

rAttributes	输出参数：
	SlAeAttributeInfo 结构的 QValueVector 的引用：
	struct SL_AE_SVC_ADAP_EXPORTS SlAeAttributeInfo
	{
	long m_nId; // 特性 ID
	QString m_strName; // 特性名称
	QString m_strDescription; // 说明
	};

返回值	错误代码

创建事件

该函数可使客户端创建任意一个事件。此时它必须遵守正确的状态机顺序，见章节6.2“报警状态机”和6.5.7“枚举数和定义”下的段落“报警状态”。

! **重要提示**

同一个报警源输出的、需应答的 HMI 报警不允许有多个实例。这是因为在某些条件下报警与事件服务无法明确 HMI 报警的出现事件的含义。
示例：
一条需应答的 HMI 报警已消失，但未经应答。如果此时报警源要输出下一个出现事件，报警与事件服务就无法明确地判定这是第一个报警实例的出现（二次出现）事件还是新实例的出现事件。

表 6-42：创建事件

long SlAeQEventSource::createEvent(SlAeEvent& rEvent)	
参数	**含义**
rEvent	需要创建的事件对象的引用。
返回值	错误代码

关于报警应答的通知

利用 Qt 信号 `eventAcknowledged` 客户端可以收到关于报警应答的通知，见章节6.5.4“通用的订阅函数”下的段落“应答报警”。
用户应答了报警源发出的一个需要应答的报警后，便发出该信号。一种应答方式是按下“诊断”操作区“报警列表”屏幕中的第 2 个垂直软键“应答报警”。另一种应答方式是调用通用的订阅函数 `acknowledgeEvent`，另见章节 6.5.4 “通用的订阅函数”。
对应报警的引用作为参数提供。函数 `createEvent` 创建的是一个状态为 `SLAE_ALARM_COMING_TOACK`、需要应答的报警。发出该信号后，报警状态自动变为 `SLAE_ALARM_CAME_ACKING`。如果在用户给出应答前报警已经被报警源设为状态 `SLAE_ALARM_CAME_GOING_TOACK`（即消失，但还没有应答），发出该信号后报警状态变为 `SLAE_ALARM_CAME_GONE_ACKING`。

表 6-43：关于事件应答的通知

void SlAeQEventSource::eventAcknowledged(const SlAeEvent& rEvent)	
参数	**含义**
rEvent	事件对象的引用。
返回值	-

关于报警删除的通知

在系统中开始某个报警删除操作后，便发送该信号。引发该操作的原因由参数 `lFlags` 指明：

- `SLAE_CANCEL_CANCEL_ALARMS`
  按下了 HMI 上的“Alarm cancel”键（“BigMac”键）。
- `SLAE_CANCEL_RECALL_ALARMS`
  按下了 HMI 上的回调键（“^”键）或向 PLC 接口发送了一个报警回调信号。

用户按下上述某个键后，相当于命令报警源删除所有 Cancel 类报警或 Recall 类报警。此时报警源必须通过函数 `createEvent` 创建对应的消失事件

SLAE_ALARM_CAME_GOING。如果在发出删除信号时引发报警的原因还是没有删除，报警源仍会首先删除报警，但之后会再次利用函数 createEvent 发出一条新报警

! **重要提示**

如果在执行报警删除操作前还同时有状态为 SLAE_ALARM_HMI_CANCEL_BTN_COMING 的 HMI 报警，报警与事件服务会首先删除这些报警，然后为每个被删除的报警发送标有 SLAE_ALARM_HMI_CANCEL_BTN 的信号 alarmCanceled。

表 6-44：关于报警删除的通知

void SlAeQEventSource::cancelAlarms(const long lFlags)	
参数	**含义**
lFlags	指出引发报警删除操作的原因： - SLAE_CANCEL_CANCEL_ALARMS - SLAE_CANCEL_RECALL_ALARMS
返回值	-

关于报警删除请求的通知

在用户选中了某个报警源发出的 HMI 报警（报警源 ID >= 10.000；对于 OEM 而言，报警源 ID >= 100.000）并按下了“诊断”操作区“报警列表”屏幕下的第 1 个垂直软键“删除 HMI 报警”后，发出该信号。另一种方法是调用通用的订阅函数 cancelAlarm，另见章节 6.5.4 “通用的订阅函数”。
用户采用上述某个方法后，相当于命令报警源删除 CancelGroup 内的报警。CancelGroup 包含了多个报警时，会一并删除这些报警。也就是说：一个或多个报警源必须通过函数 createEvent 创建一个或多个对应的消失事件 SLAE_ALARM_CAME_GOING。如果在发出该信号时引发报警的原因还是没有删除，报警源仍会首先删除报警，但之后会再次利用函数 createEvent 发出一条新报警。

表 6-45：关于报警删除请求的通知

void SlAeQEventSource::cancelAlarmGroup(const long lCancelGroup)	
参数	**含义**
lCancelGroup	指出一个 CancelGroup 内包含的、需要删除的报警。
返回值	-

关于报警自动删除的通知

在一个由报警源发出的、状态为 SLAE_ALARM_HMI_CANCEL_TIME_COMING 或者 SLAE_ALARM_HMI_CANCEL_BTN_COMING 的报警由于超时或者由于某个报警删除操作删除后，发出该信号。此时报警源无需再输出消失事件。第一个参数是一个标志位，指出引发该信号的报警删除操作。

- SLAE_ALARM_HMI_CANCEL_TIME
  报警由于超时被删除。
- SLAE_ALARM_HMI_CANCEL_BTN
  报警由于 HMI 上的“Alarm cancel”键被按下或机床操作面板 MCP 上的“RESET”键被按下而被删除。

第二个参数是对应报警的引用。

表 6-46：关于报警自动删除的通知

void SIAeQEventSource::alarmCanceled(const long lFlags, const SIAeEvent& rEvent	
参数	含义
lFlags	指出报警删除的方式： - SLAE_ALARM_HMI_CANCEL_TIME - SLAE_ALARM_HMI_CANCEL_BTN
rEvent	事件对象的引用。
返回值	-

6.5.6 SIAeEvent（单独事件）

报警与事件服务用 SIAeEvent 类的一个实例表示一个事件。

标准特性和预定义的附加特性

每个报警除了向客户端提供标识报警的报警号外，还提供其他数据，例如状态、时间戳或进程关联值。这些数据被称作特性(Attribute)，分为标准特性和附加特性(Attribute)。所有报警类别的标准特性都一样，都有自己的访问函数。而不同报警类别的附加特性有所不同，因此有统一的访问函数 `getAttribute` 和 `setAttribute`。
特性(Attribute)还根据其来源加以区分：其一方面可直接来自报警源（例如时间戳、进程关联值），另一方面还可以来源于配置（例如优先级、显示用前景色）。
附加特性可以通过配置来添加。为使客户端可以使用这些特性，订阅提供一个 Query 函数，它可返回特性的 ID、数据类型和文字说明。另外客户端还可以选择每条报警向它返回的附加特性。

报警号

Alarm-ID（报警号）用来标识与某个特定源（报警源）相关联的报警，即 Source 和 Alarm-ID 确定相应的报警文本。

! **重要提示**

同一个 Alarm-ID 可以重复在不同的报警源(Source-ID)中使用并设置不同的报警文本。即 Alarm-ID 无需在全部的报警源中保持唯一。因此报警和报警文本的（唯一性）分配只能通过 Alarm-ID 和 Source-ID 构成的组元(Tuple)实现。

表 6-47：查询报警号

quint32 getAlarmId() const	
参数	含义
返回值	报警号（报警 ID）

表 6-48：设置报警号

void setAlarmId(quint32 val)	
参数	含义
val	报警号（报警 ID）
返回值	-

为不同的报警源分别预留了一段自己的编号范围。下表列出了这些编号范围：

表 6-49：报警编号范围

序号范围	来源	描述	
000.000 – 009.999	NCK	一般报警	
010.000 – 019.999		通道报警	
020.000 – 029.999		进给轴/主轴报警	
030.000 – 039.999		功能报警	概述
040.000 – 059.999			预留
060.000 – 064.999			西门子循环报警
065.000 – 069.999			用户循环报警
070.000 – 079.999			编译循环制造商和 OEM
080.000 – 081.999			标准循环显示信息
082.000 – 082.999			Shopmill 和 CMT 循环显示信息
083.000 – 084.999			测量循环显示信息
085.000 – 089.999			用户循环显示信息
090.000 – 099.999			预留
100.000 – 129.000	HMI	系统	
130.000 – 139.000		OEM	
140.000 – 199.999		预留	
200.000 – 299.999	NCK	SINAMICS 驱动	
300.000 – 399.999		611D 驱动	
400.000 – 499.999	PLC	一般报警	
500.000 – 599.999		通道报警	
600.000 – 699.000		进给轴/主轴报警	
700.000 – 799.999		用户范围	
800.000 – 899.999		流程/图表	
(810.000 – 810.009)		系统故障显示信息	
900.000 – 999.999	NCK	611U 驱动	

报警源

源 ID 用于标识报警源。它和报警 ID 组合在一起，可以选择对应的报警文本。

表 6-50：查询报警源

quint32 getSourceId() const	
参数	**含义**
返回值	报警源（源 ID） 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”

表 6-51：设置报警源

void setSourceId(quint32 val)	
参数	**含义**
val	报警源（源 ID） 另见章节 6.5.7“枚举数和定义”下的段落“源 ID”
返回值	-

报警与事件服务中提供了几个预定义的报警源，见章节 6.5.7“枚举数和定义”。下表列出了这些报警源：

表 6-52：报警与事件服务中提供的预定义报警源

Source	源 URL	描述
0	/NCK	含驱动报警的 NCK 报警（标准机床）
1	/NCK/Channel#1/Partprogram	第 1 通道的 NCK 零件程序（标准机床）
...	...	...
10	/NCK/Channel#10/Partprogram	第 10 通道的 NCK 零件程序（标准机床）
11…49		预留
50	/PLC/DiagBuffer	NCK 诊断缓冲器中的报警与信息（标准机床）
51	/PLC/PMC	PLC 的 Alarm_S(Q)报警（标准机床）
52…99		预留
100…199	/<NCU1 的名称>/…	和标准机床一样的格式，预留给“1:N”配置中的 NCU1 使用。
200…299	/<NCU2 的名称>/…	和标准机床一样的格式，预留给“1:N”配置中的 NCU2 使用。
300…9.999	/<NCUX 的名称>/…	和标准机床一样的格式，预留给“1:N”配置中的 NCUX 使用。
10.000	/HMI	HMI 报警
10.001…99.999	/...	预留给特定应用的 HMI 报警源使用(AddOn)
100.000…199.999	/...	预留给特定应用的 HMI 报警源使用(OEM)

在“1:N”配置中，报警与事件服务收到多个 NCU 的报警时，源 URL 前面会加上 NCU 名称，比如：“/NCU_xyz/PLC/PMC”（针对 NCK 报警和 PLC 报警）。

源 URL 的句法：

源 URL ::= [/<NCU 名称>]/<主源>[/<副源>]
<NCU 名称> ::= |netnames.ini 中的 NCU 名称|<IP 地址>|
<主源> ::= |HMI|NCK|PLC|
<副源> ::= |PMC|Channel#<通道号>/Partprogram|
<通道号> ::= 零件程序发出信息所在的 NC 通道的编号，以十进制表示
<IP 地址> ::= IP 地址，以组表示，比如：“255.255.255.255”

源 ID10.000 预留给标准 HMI 报警使用。

从 10.001 起的号段供 AddOn 使用，100.000 起的号段供 OEM 使用。在该号段内，OEM 可以任意定义源 URL，但源 URL 必须以斜杠“/”开头。
在 SlAeQEventSource 类的函数 `createEvent` 中指定源 ID，可引用对应的报警源。

! **重要提示**

同一个 Alarm-ID 可以重复在不同的报警源(Source-ID)中使用并设置不同的报警文本。即 Alarm-ID 无需在全部的报警源中保持唯一。因此报警和报警文本的（唯一性）分配只能通过 Alarm-ID 和 Source-ID 构成的组元(Tuple)实现。

报警实例

在一个报警源(Source)中一个报警在同一时间可多次出现。实例 ID 用于区分这些报警实例。

! **重要提示**

例外：需要应答的 HMI 报警只能在一个时间点上出现一次。这是因为在某些条件下报警与事件服务无法明确 HMI 报警的出现事件的含义。
示例：
一条需应答的 HMI 报警已消失，但未经应答。如果此时报警源要输出下一个出现事件，报警与事件服务就无法明确地判定这是第一个报警实例的出现（二次出现）事件还是新实例的出现事件。

注

在函数 `createEvent` 中没有提供设置实例 ID 这一选项，因为报警与事件服务是自动设置实例 ID 的。

表 6-53：查询报警实例

quint32 getInstanceId() const	
参数	含义
-	-
返回值	报警实例（实例 ID）

时间标记

事件的时间戳(Timestamp)。

表 6-54：查询时间戳

QDateTime getTimestamp() const	
参数	含义
-	-
返回值	时间戳(Timestamp)

表 6-55：设置时间戳

void setTimestamp(QDateTime& rDateTime)	
参数	含义
rDateTime	时间戳(Timestamp)
返回值	-

状态

报警的状态(State)有：出现、消失和应答。该特性可显示报警是否强制需要应答。

表 6-56：查询报警状态

SlAeAlarmStateEnum getAlarmState() const	
参数	含义
-	-
返回值	报警状态(State) 另见章节 6.5.7“枚举数和定义”下的段落“报警状态”

表 6-57：设置报警状态

void setAlarmState(SlAeAlarmStateEnum val)	
参数	含义
val	报警状态(State) 另见章节 6.5.7“枚举数和定义”下的段落“报警状态”
返回值	-

质量

质量(Quality)提供关于事件可靠性的信息，比如：和 NC 的连接断开时，报警与事件服务会为每个待处理的报警生成一个质量为“差”的消失事件，因为它无法再提供报警的实际状态。

表 6-58：查询报警质量

quint16 getQuality() const	
参数	含义
-	-
返回值	报警质量 (Quality)

表 6-59：设置报警质量

void setQuality(quint16 val)	
参数	含义
val	报警质量 (Quality)
返回值	-

Cookie

Cookie 表示一个报警包含的所有事件，更确切地说，是一个报警实例包含的事件，即所有这些事件具有相同的 Cookie。换句话说，在报警进入复位状态后，Cookie 会改变，见章节 6.2“报警状态机”。
对于无需应答的报警而言，出现事件和消失事件具有相同的 Cookie。
对于需要应答的报警而言，出现事件、消失事件和应答事件具有相同的 Cookie。如果某个此类报警多次先后出现，而中途没有应答这些报警，那么所有第一个出现事件和应答前最后一个消失事件之间的事件也具有相同的 Cookie。
报警与事件服务会阻止两个前后发出的报警包含的事件具有相同的 Cookie。

表 6-60：查询 Cookie

quint32 getCookie() const	
参数	含义
-	-
返回值	Cookie

表 6-61：设置 Cookie

void setCookie(quint32 val)	
参数	含义
val	Cookie
返回值	-

类别

表示报警的类别(Category)。
并非所有的报警都支持同样的特性(Attribute)构成：比如零件程序信息就没有进程关联值。为区分报警特性，将报警划分为了不同的类别，同一类别的所有报警支持相同的特性构成。
取决于配置的不同，报警与事件服务会返回不同类别的报警。因此该服务提供了接口以查询报警类别及其特性构成。
此处 OEM 可以为现有报警类别添加新的特性或者创建新的类别。

表 6-62：查询类别

quint16 getCategory() const	
参数	含义
-	-
返回值	报警类别(Category) 另见章节 6.5.7“枚举数和定义”下的段落“类别”

表 6-63：设置类别

void setCategory(quint16 val)	
参数	含义
val	报警类别(Category) 另见章节 6.5.7“枚举数和定义”下的段落“类别”
返回值	-

预定义的附加特性

通过函数 `getAttribute` 和 `setAttribute` 可以访问下文说明的预定义附加特性。

表 6-64：查询预定义的附加特性

QVariant getAttribute(quint32 lAttributeId) const	
参数	含义
lAttributeId	特性 ID 标出了需要查询的预定义的附加特性。 另见章节 6.5.7“枚举数和定义”下的段落“预定义的附加特性”
返回值	报警类别(Category)

表 6-65：设置预定义的附加特性

bool setAttribute(quint32 lAttributeId, QVariant attribute)	
参数	含义
lAttributeId	特性 ID 标出了需要设置的预定义的附加特性。 另见章节 6.5.7“枚举数和定义”下的段落“预定义的附加特性”
attribute	预定义附加特性的值。
返回值	-

6.5.7 枚举数和定义

源 ID

通过函数 `setAttribute` 和 `getAttribute` 访问 SlAeQEvent 预定义附加特性时，有以下常数：

`SLAE_HMI_SOURCE_ID`
HMI 报警源的 ID（和 SLAE_SOURCE_ID_HMI 一致）

`SLAE_SOURCE_ID_HMI`
HMI 报警源的 ID（和 SLAE_HMI_SOURCE_ID 一致）

`SLAE_SOURCE_ID_NCK`
标准机床中 NCK 报警源的 ID（含所有驱动报警）

`SLAE_SOURCE_ID_NCK_CHAN_01_PPM`
标准机床通道 1 中 NC 零件信息报警源的 ID

`SLAE_SOURCE_ID_NCK_CHAN_02_PPM`
标准机床通道 2 中 NC 零件信息报警源的 ID

`SLAE_SOURCE_ID_NCK_CHAN_03_PPM`
标准机床通道 3 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_04_PPM
标准机床通道 4 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_05_PPM
标准机床通道 5 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_06_PPM
标准机床通道 6 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_07_PPM
标准机床通道 7 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_08_PPM
标准机床通道 8 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_09_PPM
标准机床通道 9 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_NCK_CHAN_10_PPM
标准机床通道 10 中 NC 零件信息报警源的 ID

SLAE_SOURCE_ID_PLC_DIAG_BUFFER
标准机床中 PLC 诊断缓冲器报警源的 ID

SLAE_SOURCE_ID_PLC_PMC
标准机床中 PLC MQS 缓冲器报警源的 ID

类别

属于一个类别的所有报警具有相同的特性组。所有报警类别的标准特性都是一样的。但预定义的附加特性是不一样的。

SLAE_EV_CAT_SINUMERIK
Sinumerik 类

SLAE_EV_CAT_SINUMERIK_HMI
HMI 报警类

SLAE_EV_CAT_SINUMERIK_PARTPROGMSG
零件程序信息类

预定义的附加特性

通过函数 setAttribute 和 getAttribute 访问 SlAeQEvent 预定义附加特性时，有以下常数：

SLAE_EV_ATTR_SEVERITY
报警的优先级。优先级(Severity)通过定义报警特性来指定。优先级可以成为报警排序的依据。

SLAE_EV_ATTR_MSGTEXT
由 NC 发出的报警的文本以及参数。

SLAE_EV_ATTR_PARAM01 到 SLAE_EV_ATTR_PARAM10

报警的参数 1 到 10（关联值、进程值）。只要能从控制系统或配置文件确定出数据类型，参数就采用该数据类型，否则作为字符串传送。

SLAE_EV_ATTR_DISPLOC
显示位置(DisplayLocation)决定了报警在哪个位置上显示，是在标准显示栏（诊断的标题栏）还是在消息框中显示。

SLAE_EV_ATTR_HELPFILENAME
HelpFileName 可对本地或网络服务器上的 HTML 文件进行本地化，然后显示该文件。

SLAE_EV_ATTR_TYPE
定义报警特性时的排序类别(Type)，为确保与 OPC 兼容而使用。

SLAE_EV_ATTR_CANCELGROUP
具有相同 CancelGroup（删除组）的报警构成一个总删除组。通过删除一个报警可自动将该组的所有报警都删除。该删除操作是由报警与事件服务执行的。只有报警是无状态报警时，CancelGroup 才起作用。

SLAE_EV_ATTR_CLEARINFO
ClearInfo（删除标准）同样表示一个报警组，此处的值用来说明通过哪种操作可删除一个报警（以及该组的所有报警），例如“NC 复位”。与 CancelGroup 相反的是，这些报警的删除过程不是由报警与事件服务执行的，而是直接在报警源（例如 NCK）中进行，即 NCK 作为报警源会为每个报警发送一个消失事件。下表列出了 ClearInfo 的预定义值。

表 6-66：删除标准一览

删除标准 (ClearInfo)	报警源 (Source)	描述
0	HMI	通过 HMI 删除的报警。
1	NCK	通过 NCU 重新上电删除的报警。
2		通过 NCU 硬件复位进行删除的条件。
3		通过向 NCU 发送“Cancel”命令进行删除的条件。
4		通过 NCK 本身进行删除的条件。
5		通过 NCU 的“NC 启动”命令进行删除的条件。
6		通过复位运行方式组(BAG)进行删除的条件。
7		通过 NCU 的“NC 复位”命令进行删除的条件。
8	PLC	FB15 的 PLC 信息（基本程序）
9		FB15 的 PLC 报警（基本程序）
10		通过“回调”键[^]删除的 HMI 诊断报警。
11		预留
12		PLC(SFC17/18)的 S7-PDiag、S7-Graph、S7-HiGraph 报警或其他的 Alarm_S(Q)报警，报警状态为“未应答”
13		PLC(SFC17/18)的 S7-PDiag、S7-Graph、S7-HiGraph 报警或其他的 Alarm_S(Q)报警，报警状态为“已应答”(“acknowledged”)
14	驱动(NCK)	通过 NCK 发出的驱动报警
15		零件程序信息

报警状态

下面是一些指出报警或事件状态的基本标志位：

SLAE_ALARM_UNKNOWN_STATE
未知状态。

SLAE_ALARM_STATE_ACTIVE
报警有效。

SLAE_ALARM_STATE_ACKED
报警被应答。

SLAE_ALARM_STATE_ENABLED
报警被使能。

SLAE_ALARM_CHANGE_ACTIVE
“有效”标志位已更改。

SLAE_ALARM_CHANGE_ACKED
“应答”标志位已更改。

SLAE_ALARM_CHANGE_ENABLED
“使能”标志位已更改。
SLAE_ALARM_ACK_REQUIRED
报警需要应答。

SLAE_ALARM_HMI_CANCEL_BTN

通过 HMI 的 Cancel 键（“BigMac”键）自动删除 HMI 报警。

SLAE_ALARM_HMI_CANCEL_TIME
通过超时（Timeout）自动删除 HMI 报警。

无需应答的报警的状态

SLAE_ALARM_COMING
无需应答的报警刚刚出现。
(SLAE_ALARM_STATE_ENABLED + SLAE_ALARM_STATE_ACTIVE + SLAE_ALARM_CHANGE_ACTIVE)

SLAE_ALARM_CAME_GOING
无需应答的报警刚刚出现并消失。
(SLAE_ALARM_STATE_ENABLED + SLAE_ALARM_CHANGE_ACTIVE)

可通过Cancel键（“BigMac”键）删除的无需应答的HMI报警的状态

SLAE_ALARM_HMI_CANCEL_BTN_COMING
可通过 Cancel 键（“BigMac”键）删除的无需应答的 HMI 报警刚刚出现。
(SLAE_ALARM_HMI_CANCEL_BTN + SLAE_ALARM_COMING)

SLAE_ALARM_HMI_CANCEL_BTN_CAME_GOING
可通过 Cancel 键（“BigMac”键）删除的无需应答的 HMI 报警刚刚消失。
(SLAE_ALARM_HMI_CANCEL_BTN + SLAE_ALARM_CAME_GOING)

可通过超时(Timeout)删除的无需应答的HMI报警的状态

SLAE_ALARM_HMI_CANCEL_TIME_COMING
可通过超时(Timeout)删除的无需应答的 HMI 报警刚刚出现。
(SLAE_ALARM_HMI_CANCEL_TIME + SLAE_ALARM_COMING)

SLAE_ALARM_HMI_CANCEL_TIME_CAME_GOING
可通过超时(Timeout)删除的无需应答的 HMI 报警刚刚消失。
(SLAE_ALARM_HMI_CANCEL_TIME + SLAE_ALARM_CAME_GOING)

需应答的报警的状态

SLAE_ALARM_COMING_TOACK
需应答的报警刚刚出现且仍必须应答。
(SLAE_ALARM_ACK_REQUIRED + SLAE_ALARM_COMING)

SLAE_ALARM_CAME_GOING_TOACK
需应答的报警刚刚出现并消失，但仍必须应答。
(SLAE_ALARM_ACK_REQUIRED + SLAE_ALARM_CAME_GOING)

SLAE_ALARM_CAME_GONE_ACKING
需应答的报警出现，消失并刚刚被应答。
(SLAE_ALARM_ACK_REQUIRED + SLAE_ALARM_STATE_ENABLED + SLAE_ALARM_STATE_ACKED + SLAE_ALARM_CHANGE_ACKED)

SLAE_ALARM_CAME_ACKING
需应答的报警出现并刚刚被应答。
(SLAE_ALARM_ACK_REQUIRED + SLAE_ALARM_STATE_ENABLED + SLAE_ALARM_STATE_ACTIVE + SLAE_ALARM_STATE_ACKED + SLAE_ALARM_CHANGE_ACKED)

SLAE_ALARM_CAME_ACKED_GOING
需应答的报警出现，已被应答并刚刚消失。
(SLAE_ALARM_ACK_REQUIRED + SLAE_ALARM_STATE_ENABLED + SLAE_ALARM_STATE_ACKED + SLAE_ALARM_CHANGE_ACTIVE)

筛选

在使用函数 setFilter 筛选订阅时有以下常数：

SLAE_FILTER_NO_FILTER
无筛选

SLAE_FILTER_MESSAGES_ONLY
仅允许信息

SLAE_FILTER_ALARMS_ONLY
仅允许报警

报警的删除方式

在使用函数 cancelAlarms 删除报警时有以下常数：

SLAE_CANCEL_CANCEL_ALARMS
删除 HMI Cancel 型和 NCK Cancel 型报警

SLAE_CANCEL_RECALL_ALARMS
仅删除 HMI 报警

另见章节 6.5.5“报警源(SlAeQEventSource)”的段落“关于报警删除的通知”

排序

在使用函数 setSorting 对 SlAeQAlarmPtrList 进行排序时有以下常数：

SLAE_SORTBY_SEVERITY
优先级

SLAE_SORTBY_TIMESTAMP
时间标记

SLAE_SORTBY_SEQUENTIAL
出现的顺序

SLAE_ORDER_ASCENDING
升序

SLAE_ORDER_DESCENDING
降序

7

7 目录及文件服务

本章主要内容

本章介绍文件服务（File service）的接口。这适用于文件或目录操作。

7.1 引言

7.1.1 类模型

概述

文件服务的类模型主要由以下的类组成：

- SlQFileSvc

SlQFileSvc 类

借助 SlQFileSvc 对象可实现对文件和目录的操作。操作包括：创建、复制、移动和删除文件和目录，以及查看目录的内容。

此外还提供用于 Sinumerik 专用任务的函数。例如处理访问权限和选择要执行的零件程序。

很多调用既可以采取同步方式，也可以采取异步方式。其中，同步调用会阻塞调用所在的线程，直到任务完成。而在异步调用中，SlQFileSvc 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

SlQFileSvc 类的所有实例共用文件服务的一个接口，但是每个实例都实现一个回调接口，以便不受其他实例影响地接收结果。因此 SlQFileSvc 类在这一方面影响不是很大。

注

SlQFileSvc 对象只能应用在 Qt 主线程中。

7.1.2 术语解释

同步调用

同步调用在任务执行完后才会返回，即当前线程会被一直阻塞。这会干扰事件处理，例如在同步调用时阻止输入和显示。因此比较耗时的调用应进行异步调用。

异步调用

只要任务被发送给操作记录服务，
异步调用就返回。这尤其意味着，所提供的故障代码不代表任务是否成功完成，而只是任务被成功发送的回复。例如当未正确提供调用参数时，则会出错。真正的任务状态在回调文件服务时提供（信号与槽机制）。

文件和目录

可向每个可寻址的文件和目录询问以下信息(Attribute)。

表 7-1：文件和目录的特性(Attribute)

特征值	含义
逻辑路径	文件或目录的逻辑路径。 （示例："//NC/MPF.DIR/TEST.MPF"）
实际路径	文件或目录的实际路径。 （示例："/_N_MPF_DIR/_N_TEST_MPF"）
数据格式	表明对象是一个文件还是目录。
主名称	文件主名称。（示例："TEST.MPF"）
访问标记	文件/目录的存取权限。（示例： "77777")
信号	以字节为单位的文件大小。 对于目录，该值为-1。
时间标记	最后一次修改的时间戳。
简明文本	简明文本标识（只针对 NC 文件系统）

保存文件有以下方式：

1. 在 CF 卡上，例如“/card/siemens/sinumerik/mpf.dir/test.mpf”
2. 在 NC 的数据结构中，例如“//NC/mpf.dir/test.mpf”
3. 向 TCU-USB 驱动器上，例如“//ACTTCU/FRONT/mpf.dir/test.mpf”
4. 在网络驱动器上，例如“//PCname/mpf.dir/test.mpf”

访问授权

对文件和目录有八个访问级别。

表 7-2：八个访问级别

访问级	获取方式	用户组
S0	系统口令	西门子
S1	机床制造商口令	机床制造商
S2	售后服务口令	调试人员，售后服务 （机床制造商）
S3	最终用户口令	特许最终用户（自行售后服务）
S4	钥匙开关位置 3	编程人员
S5	钥匙开关位置 2	受过培训的操作员
S6	钥匙开关位置 1	操作人员
S7	钥匙开关位置 0	半熟练的操作人员 （NC 启动/NC 停止，MSTT）

S0 代表最高权限的访问级别，S7 代表最低权限的访问级别。

访问级别可通过实际的钥匙开关位置或输入口令来释放。

访问权限

可以分配以下权限：

READ --> 读出
WRITE --> 写入
EXECUTE --> 执行
SHOW --> 查看
DELETE --> 删除

访问标记

每个文件和每个目录都设有一个访问标记，它确定了从哪个访问级别起可处理文件和目录。访问标记是一个五位数的数字。

表 7-3：访问标记

第 1 位	第 2 位	第 3 位	第 4 位	第 5 位
READ	WRITE	EXECUTE	SHOW	DELETE

示例：
文件的访问标记为 74775

1. 访问级别 S0 到 S4 允许执行写操作
2. 访问级别 S0 到 S5 允许执行删除操作
3. S0 到 S7 的所有访问级别都允许执行读、执行和查看操作

Powerline 升级版的说明

在 HMI 高级版编程包的升级版中，文件服务不再使用函数 activate/passivate（比如用于激活/取消激活零件程序）。取而代之的是函数“moveFile/moveFileEx”或“copyFile/copyFileEx”。

此处需要注意的是，用户必须自行确保数据是一致的。

示例：
应用程序在 CF 卡的目录“/card/tmp/”下生成一个零件程序“Kreis.mpf”。现在利用“copyFile”可将将该文件复制到
NC 目录“/NC/_N_MPF_DIR/_N_KREIS_MPF”下。源文件“/card/tmp/Kreis.mpf”不会被删除。用相同的路径重新激活或禁用会改写另一个零件程序。为此用户必须确保应用程序的工作方式符合其要求。

7.2 分步示例

概述

以下章节将分步介绍文件服务的不同应用区域。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例。

以下为这些示例和所有其他示例一览。

表 7-4：示例一览

应用	方式	示例	章节
查看目录内容	同步	slexfilesynclistfolder	7.2.2
	异步	slexfileasynclistfolder	7.2.3
创建、复制和删除文件	同步	slexfilesynccopyremove	7.2.4
	异步	slexfileasynccopyremove	7.2.5
复制文件，弹出确认提示框		slexfileasynccopyremoveask	
查看文件或目录的特性(Attribute)		slexfileattributes	7.2.6
转换路径名称		slexfileconvertpath	7.2.7

此处只举例说明或描述与文件服务相关的内容。源文件的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

注

如果在示例程序中选择的是 CF 卡上的路径，则示例程序只能在嵌入式 Linux 系统上运行。无法从 Windows 访问 CF 卡。

开发环境

建立开发环境所需的步骤在章节 5.3 中说明。

7.2.1 准备

概述

满足以下前提后，才能从自定义的项目或类中调用文件服务。这些条件同样也针对下文中的所有“分步示例”。

检查库

在项目设置中检查库 `slfsfilesvcadapter.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `slfsfilesvcadapter.lib`

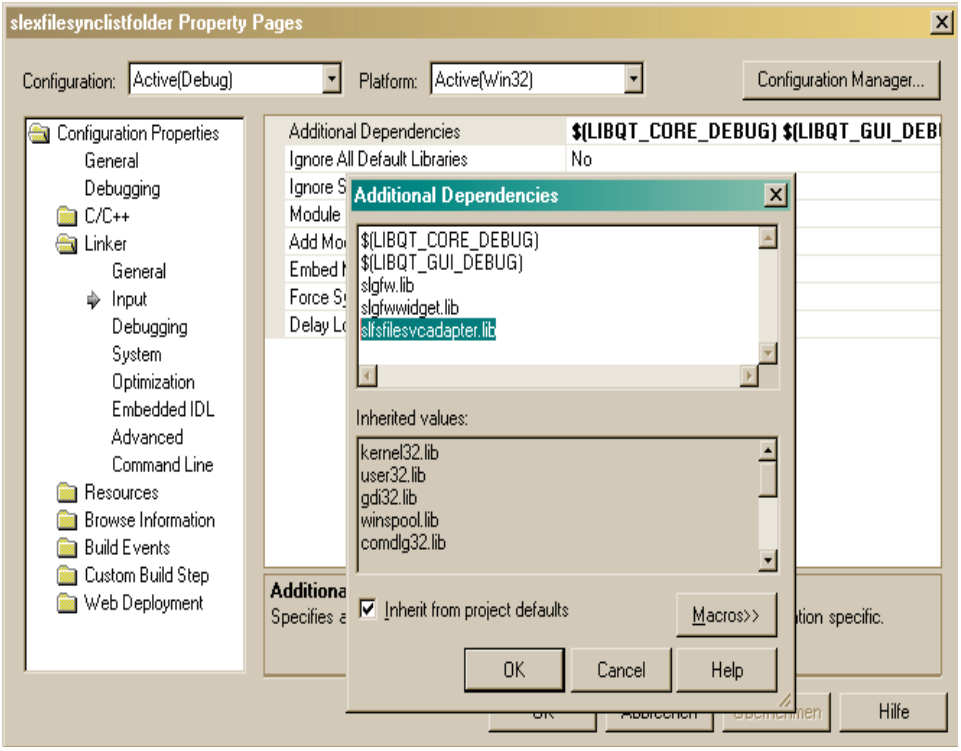


图 7-1:库 slfsfilesvcadapter.lib

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lslfsfilesvcadapter
```

头文件

要使用文件服务的类需要加入头文件"slqfilesvc.h"，配置条目为：

```
#include "slqfilesvc.h"
```

创建 SlQFileSvc 对象

访问文件服务的接口需要使用 SlQFileSvc 类的对象。这些对象可作为私有的成员变量创建：

```
SlQFileSvc m_fileServer;
```

注

SlQFileSvc 类的对象只有少数的实例数据，在创建和删除时费时较短。因此 SlQFileSvc 对象也可以在栈上创建。

初始化 SlQFileSvc

在使用 SlQFileSvc 类的各个函数前，首先要初始化该类。为此要调用函数 `init()`。如果 SlQFileSvc 对象是作为私有成员变量创建的，最好在
构造函数中进行初始化：

```
m_fileServer.init();
```

释放资源

不再需要 SIQFileSvc 对象时，需要调用函数 `fini()`。这样可以再次释放原先被占用的资源。如果 SIQFileSvc 对象是作为私有成员变量创建的，最好在析构函数中释放资源：

```
m_fileServer.fini();
```

7.2.2 查看目录内容（同步）

概述

下面的示例分步介绍了如何同步读取目录的内容。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExFileSyncListFolder”。

在示例中可给出路径，路径通过软键“list folder”显示。状态栏显示调用成功。

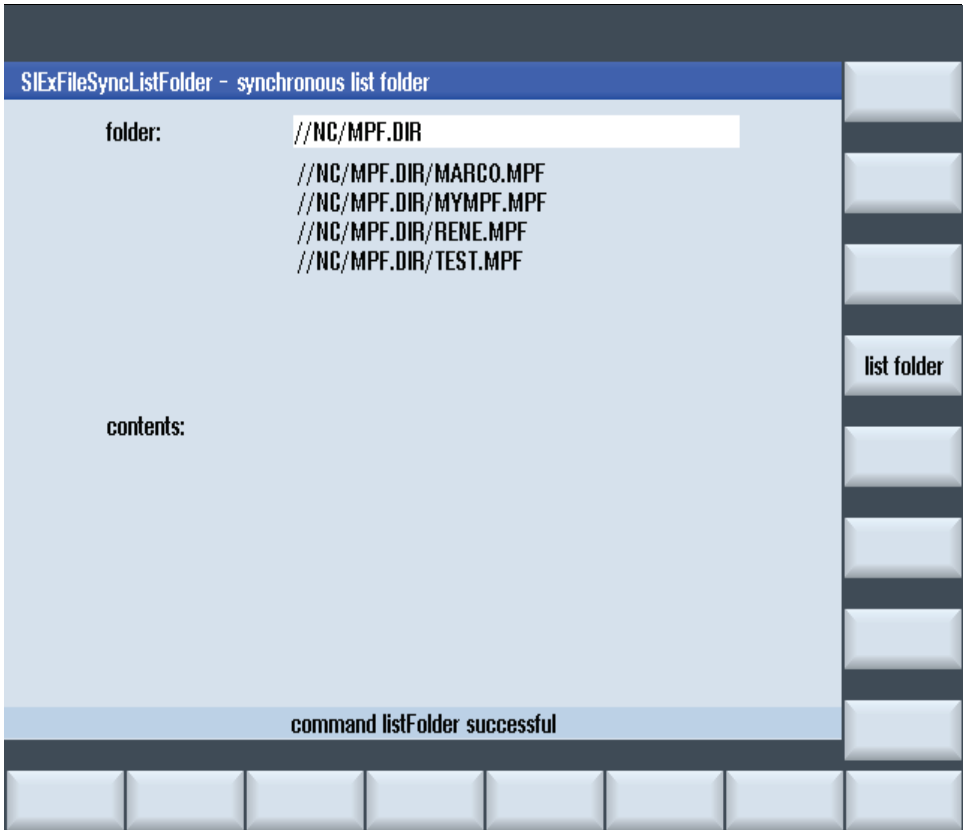


图 7-2:示例 SIExFileSyncListFolder

务必要满足章节 7.2.1“准备”列出的前提条件。

第 1 步

创建 NodeInfoArray。它是一个来自 NodeInfo 对象的数组，
NodeInfo 包含了文件或目录的特性。

```
NodeInfoArray nodes;
```

第 2 步

向 NodeInfoArray 填入 NC 零件程序目录的内容。

```
long lRetVal = m_fileServer.listFolder("//NC/MPF.DIR", nodes);
```

第 3 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
                                                                    // 插入故障处理
}
else
{
                                                    //nodes 包含了特性，用于后续处理
}
```

第 4 步

处理 NodeInfoArray 的内容。

```
for ( int nIndex = 0 ; nIndex < nodes.count() ; nIndex++ )
{
      // 比如：将逻辑路径复制到一个 QString 中
      QString szItem = nodes[nIndex].strPath;
      ...
}
```

7.2.3 查看目录内容（异步）

概述

下面的示例分步介绍了如何异步读取目录的内容。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExFileAsyncListFolder”。该示例程序的结构与 7.2.2 章描述的同步查看目录内容的示例相同。

务必要满足章节 7.2.1“准备”列出的前提条件。

第 1 步

创建一个 long 型私有成员变量。该变量用于之后标识异步调用。

```
long m_lRequestId;
```

第 2 步

任务的执行结果通过信号通知文件服务。该信号由以下函数（槽）接收。函数的定义在第 5 步(listCompletedSlot)中变得清晰明确。

```
private slots:
      void listCompletedSlot(long, NodeInfoArray&, long);
```

第 3 步

现在将成员函数（槽）与信号 listCompleted 关联在一起。

```
QObject::connect(&m fileServer,
                     SIGNAL(listCompleted(long, NodeInfoArray&, long)),
                     this,
                     SLOT(listCompletedSlot(long, NodeInfoArray&, long)));
```

第 4 步

向文件服务传送任务：读出 NC 目录“零件程序”的内容。

```
long lRetVal = m_fileServer.listFolderEx("//NC/MPF.DIR", m_lRequestId);
```

第 5 步

调用立即返回。任务的执行结果位于槽 `listCompletedSlot` 中。需要检查它是否是正确的任务时，可以对比 `m_lRequestId` 与槽 `listCompletedSlot` 传递的 `lRequestId`。

```
void SlExFileAsyncListFolderForm::listCompletedSlot(
                                                long lRequestId,
                                                NodeInfoArray& rNodes,
                                                long lRetVal)
{
      if ( m lRequestId == lRequestId )
      {
            if( 0 != lRetVal )
            {
                  // 插入故障处理
            }
            else
            {
                  //rNodes 包含了特性，用于
                  // 后续处理
            }
      }
}
```

第 6 步

现在您可以在 `listCompletedSlot` 中处理 NodeInfoArray 的内容。

```
for ( int nIndex = 0 ; nIndex < rNodes.count() ; nIndex++ )
{
            // 比如：将逻辑路径复制到一个 QString 中
            QString szItem = rNodes[nIndex].strPath;
            ...
}
```

7.2.4 创建、复制和删除文件（同步）

概述

下面的示例分步介绍了如何同步创建文件，同步复制以及再将其同步删除。
SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExFileSyncCopyRemove”。

示例中编辑的是一个零件程序。程序在按下软键“create source”后生成。通过“copy source”可将该文件复制到指定的目标路径下。然后可以通过软键“remove source”或“remove destination”再将这两个文件删除。状态栏显示调用成功。

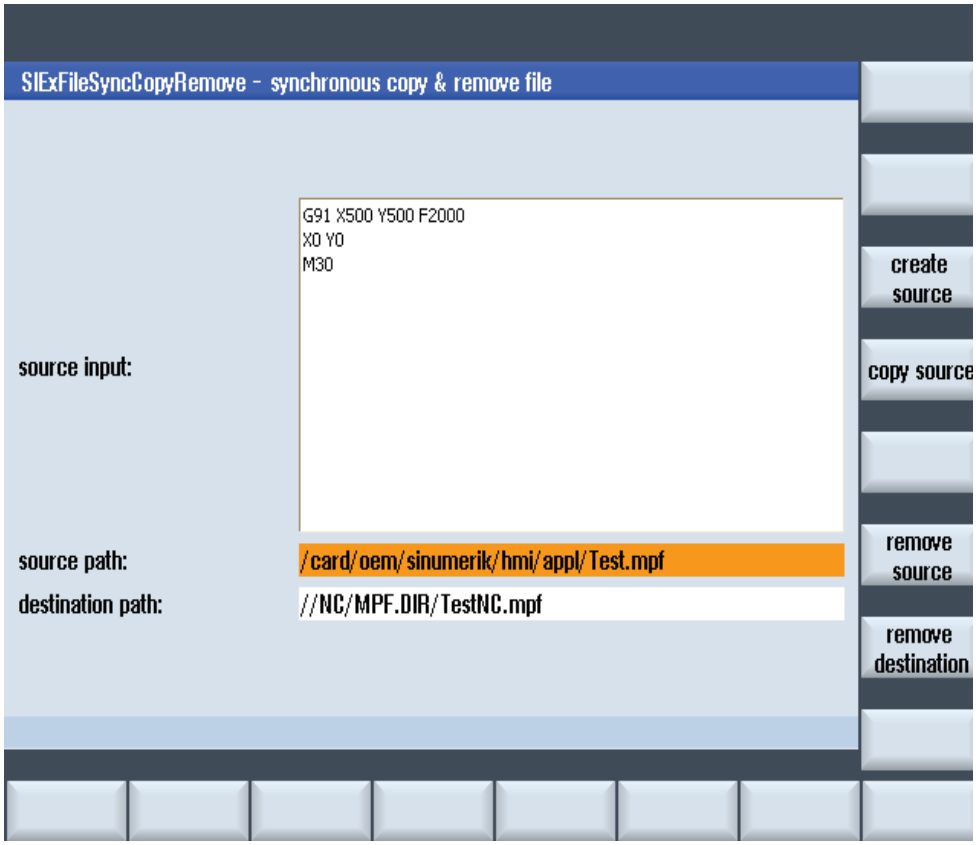


图 7-3:示例 SIExFileSyncCopyRemove

务必要满足章节 7.2.1“准备”列出的前提条件。

第 1 步

首先在路径“/card/oem/sinumerik/hmi/appl”下创建零件程序 `Test.mpf`。

```
long lRetVal = m_fileServer.createFile(
            "/card/oem/sinumerik/hmi/appl/Test.mpf", true);
```

第 2 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
      // 插入故障处理
}
```

第 3 步

使用 Qt 类 QFile 向零件程序填入内容。

```
QFile file("/card/oem/sinumerik/hmi/appl/Test.mpf");
if ( true == file.open(QIODevice::WriteOnly | QIODevice::Text) )
{
      QTextStream out(&file);
      out << QString("M30") << endl;
      file.close();
}
```

第 4 步

将零件程序从“/card/oem/sinumerik/hmi/appl/Test.mpf”复制到路径“//NC/MPF.DIR/TestNC.mpf”下。

```
long lRetVal = m_fileServer.copyFile(
            "/card/oem/sinumerik/hmi/appl/Test.mpf",
            "//NC/MPF.DIR/TestNC.mpf", true);
```

第 5 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
      // 插入故障处理
}
```

第 6 步

再次删除零件程序//NC/MPF.DIR/TestNC.mpf。

```
long lRetVal = m_fileServer.remove("//NC/MPF.DIR/TestNC.mpf");
```

第 7 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
      // 插入故障处理
}
```

7.2.5 创建、复制和删除文件（异步）

概述

下面的示例分步介绍了如何异步创建文件，异步复制以及再将其异步删除。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExFileAsyncCopyRemove”。该示例程序的结构与 7.2.4 章描述的同步创建、复制和删除文件的示例相同。

务必要满足章节 7.2.1“准备”列出的前提条件。

第 1 步

创建两个 long 型私有成员变量。这两个变量用于之后标识异步调用。

```
long m_lRequestId_Copy;
long m_lRequestId_Remove;
```

第 2 步

任务的执行结果通过信号通知文件服务。该信号由以下函数（槽）接收。函数的定义在第 8 步(`completedSlot`)中变得清晰明确。

```
private slots:
   void completedSlot(long nRequestId, long errCode);
```

第 3 步

现在将成员函数（槽）与信号 `completed` 关联在一起。

```
QObject::connect(&m fileServer,
                  SIGNAL(completed(long, long)),
                  this,
                  SLOT(completedSlot(long, long)));
```

第 4 步

首先在路径“`/card/oem/sinumerik/hmi/appl`”下创建零件程序 `Test.mpf`。

```
long lRetVal = m_fileServer.createFile(
                  "/card/oem/sinumerik/hmi/appl/Test.mpf", true);
```

注

createFile 没有异步调用。

第 5 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
   // 插入故障处理
}
```

第 6 步

使用 Qt 类 QFile 向零件程序填入内容。

```
QFile file("/card/oem/sinumerik/hmi/appl/Test.mpf");
if ( true == file.open(QIODevice::WriteOnly | QIODevice::Text) )
{
   QTextStream out(&file);
   out << QString("M30") << endl;
   file.close();
}
```

第 7 步

向文件服务传送任务：将零件程序“/card/oem/sinumerik/hmi/appl/Test.mpf”复制到路径“//NC/MPF.DIR/TestNC.mpf”下。

```
long lRetVal = m_fileServer.copyFileEx(
                        "/card/oem/sinumerik/hmi/appl/Test.mpf",
                        "//NC/MPF.DIR/TestNC.mpf",
                        m_lRequestId_Copy, true);
```

第 8 步

调用立即返回。执行结果位于槽 completedSlot 中。需要检查它是否是正确的任务时，可以对比 m_lRequestId_Copy 与槽 completedSlot 传递的 lRequestId。

```
void SlExFileAsyncCopyRemoveForm::completedSlot(long lRequestId,
                                                long lRetVal)
{
   if ( lRequestId == m_lRequestId_Copy )
   {
            if( 0 != lRetVal )
            {
                  // 插入故障处理
            }
   }
}
```

第 9 步

向文件服务传送任务：删除零件程序“//NC/MPF.DIR/TestNC.mpf”。

```
long lRetVal = m_fileServer.removeEx("//NC/MPF.DIR/TestNC.mpf",
                                     m_lRequestId_Remove);
```

第 10 步

调用立即返回。执行结果位于槽 completedSlot 中。需要检查它是否是正确的任务时，可以对比 m_lRequestId_Remove 与槽 completedSlot 传递的 lRequestId（见第 8 步）。

7.2.6 查看文件或目录的特性(Attribute)

一览

下面的示例分步介绍了如何读取文件或目录的特性。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExFileAttributes”。

在该示例中可以按下软键“get Attributes”来查看文件或目录的特性。状态栏显示调用成功。

图 7-4:示例 SIExFileAttributes

务必要满足章节 7.2.1“准备”列出的前提条件。

第 1 步

创建一条 NodeInfo。它是文件服务的一个对象，包含了文件或目录的特性。

```
NodeInfo node;
```

第 2 步

向 NodeInfo 填入 NC 零件程序目录中文件 `Test.mpf` 的特性。

```
long lRetVal = m_fileServer.getAttributes("//NC/MPF.DIR/TEST.MPF", node);
```

第 3 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
      // 插入故障处理
}
else
{
      //node 包含了特性，用于后续处理
}
```

第 4 步

现在可以处理 NodeInfo 的内容。

```
QString strRealPath = node.strRealPath;
bool bFolder = node.bFolder;
QString strBaseName = node.strBaseName;
QString strAccessMask = node.strAccessMask;
long nSize = node.nSize;
```

第 5 步

按以下方式将上次更改的时间戳写入 QString 中。

```
QDateTime date;
date.setTime_t(node.nLastWriteTimestamp);
QString szDate = date.toString("dd.MM.yy hh:mm");
```

7.2.7 转换路径名称

概述

下面的示例分步介绍了如何将在实际路径名和逻辑路径名之间转换。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExFileConvertPath”。

在该示例中，可按下软键“real -> logical”将实际路径转换为对应的逻辑路径。按下“logical -> real”将逻辑路径转换为实际路径。状态栏显示调用成功。

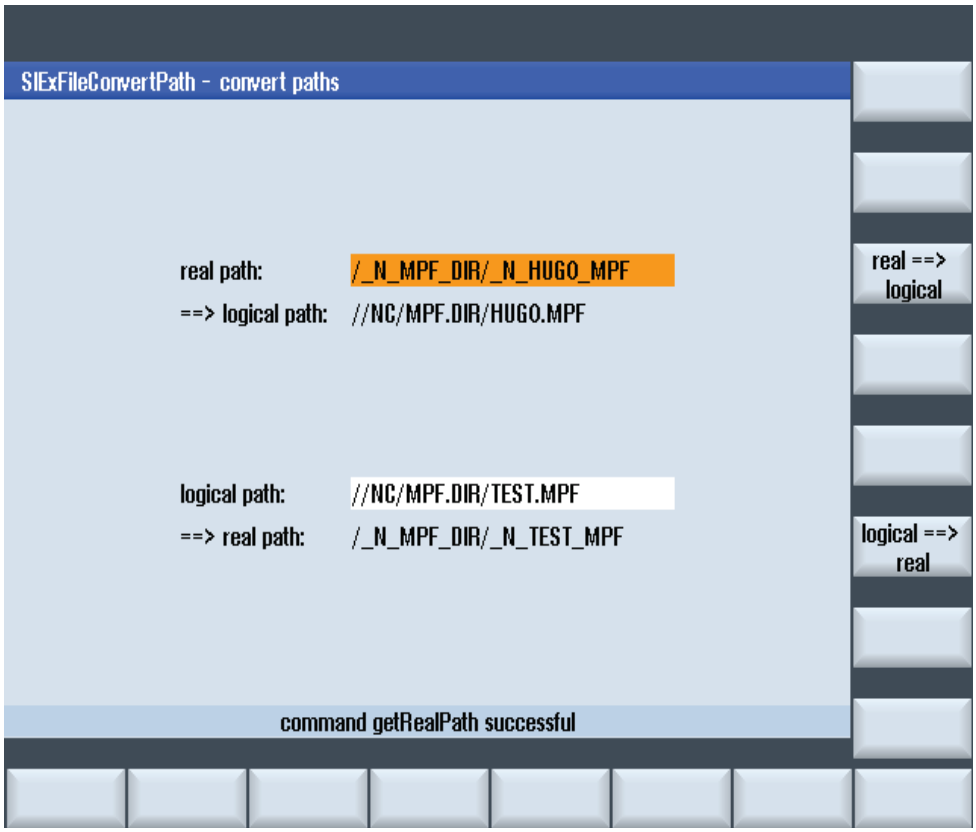


图 7-5:示例 SIExFileConvertPath

务必要满足章节 7.2.1“准备”列出的前提条件。

第 1 步

作为返回值创建一个 QString。

```
QString szLogicalPath = QString::null;
```

第 2 步

将实际路径`/_N_MPF_DIR/_N_HUGO_MPF`转换为逻辑路径(`//NC/MPF.DIR/HUGO.MPF`)。

```
long lRetVal = m_fileServer.getLogicalPath("/_N_MPF_DIR/_N_HUGO_MPF",
                                            szLogicalPath);
```

第 3 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
      // 插入故障处理
}
else
{
      // szLogicalPath 包含了逻辑路径
}
```

第 4 步

逻辑路径到实际路径的转换与此类似。

```
QString szRealPath = QString::null;
long lRetVal = m_fileServer.getRealPath("//NC/MPF.DIR/HUGO.MPF",
                                          szRealPath);
```

第 5 步

查看任务的执行状态。

```
if( 0 != lRetVal )
{
      // 插入故障处理
}
else
{
      // szRealPath 包含了实际路径
}
```

7.3 SIQFileSvc 引用

7.3.1 定义

概述

借助 SIQFileSvc 对象可实现对文件和目录的操作。操作包括：创建、复制、移动和删除文件和目录，以及查看目录的内容。

此外还提供用于 Sinumerik 专用任务的函数。例如处理访问权限和选择要执行的零件程序。

很多调用既可以采取同步方式，也可以采取异步方式。其中，同步调用会阻塞调用所在的线程，直到任务完成。而在异步调用中，SIQFileSvc 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

SIQFileSvc 类的所有实例共用文件服务的一个接口，但是每个实例都实现一个回调接口，以便不受其他实例影响地接收结果。因此 SIQFileSvc 类在这一方面影响不是很大。

路径结构

以下示例显示了逻辑路径的结构，逻辑路径应用作文件服务中函数的输入参数。

表 7-5：逻辑路径

逻辑路径	描述
//NC/mpf.dir/test.mpf	NC 路径
/card/oem/sinumerik/hmi/appl/test.mpf	Linux 路径
//ACTTCU/FRONT/mpf.dir/test.mpf //ACTTCU/X203/test.mpf	本地 TCU-USB 驱动器
//PCname/mpf.dir/test.mpf	Linux 网络驱动器
//192.168.10.17/guest/test.mpf	WIndows 网络驱动器
//NC/act.dir	NC 主动文件系统

注

使用网络驱动器时，应先在“调试”操作区域中配置相应的驱动器。

注

调用 SIQFileSvc 接口时只能使用逻辑路径。不能使用实际路径（例如“/_N_MPF_DIR/_N_TEST_MPF”）。为此可使用函数 *getLogicalPath*，它可将实际路径转换为逻辑路径。

1:N 访问的路径结构

如果存在 1:N 配置便可以访问特定的控制系统。 可通过前置前缀进行访问：

*//NCU/**NCU-Name**/NC/Pfad*

*NCU-名称*符合 *mmc.ini* 中配置的 NCU 名称：

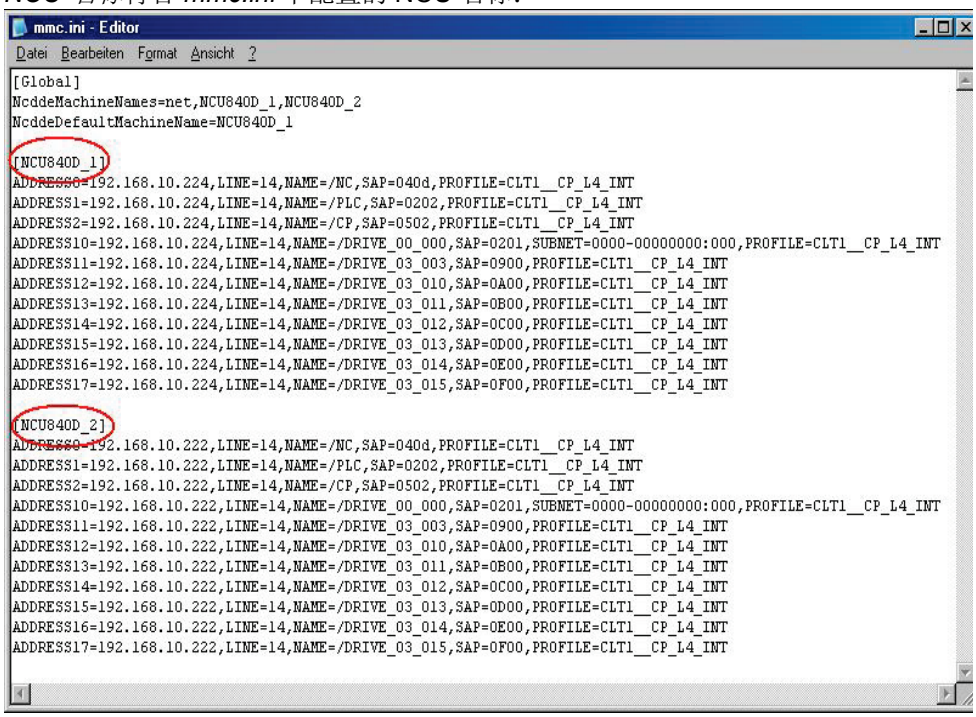


图 7-6：mmc.ini 中的 NCU 名称

表 7-6： 逻辑路径（1:N 访问）

变量路径	描述
//NCU/NCU_1/NC/MPF.DIR/MPF_1.MPF	NCU_1 零件程序
//NCU/NCU_2/NC/SPF.DIR/MySPF.SPF	NCU_2 子程序
//NCU/MyNCU/NC/MPF.DIR	MyNCU 零件程序目录

7.3.2 初始化和终止函数

初始化(init)

初始化 SIQFileSvc 对象。该函数作为第一个函数调用。

表 7-7：init

long SIQFileSvc::init();	
参数	**含义**
返回值	函数的执行状态

示例：
在每个“分步示例”中使用。

终止(fini)

释放 SIQFileSvc 的资源。该函数作为最后一个函数调用。

表 7-8：fini

long SIQFileSvc::fini();	
参数	含义
返回值	函数的执行状态

示例：
在每个“分步示例”中使用。

7.3.3 管理函数

此函数可列出目录的内容。结果作为一个 NodeInfoArray 数组返回，它和数据类型 QVector<NodeInfo>一致。NodeInfo 描述了一个文件或目录。

目录较大时，调用会持续较长时间。此时应使用异步调用(listFolderEx)，以避免阻塞调用所在的线程。

表 7-9：listFolder

long SIQFileSvc::listFolder(const QString& strFolderPath, NodeInfoArray& rNodes);	
参数	含义
strFolderPath	待查看的目录。
rNodes	目录的内容。
返回值	显示任务的执行状态。

在异步调用中会在调用后返回一个句柄，用于之后标识调用。

该函数发送以下信号：progress、listCompleted，有时也发送 canceled。

表 7-10：listFolderEx

long SIQFileSvc::listFolderEx(const QString& strFolderPath, long& nRequestId);	
参数	含义
strFolderPath	待查看的目录。
nRequestId	用于标识异步调用的句柄(handle)。
返回值	函数的执行状态。

示例：
参见章节 7.2.2（同步）和 7.2.3（异步）
（“分步示例”）

创建文件(createFile)

该函数可用于创建一份文件。如果在目标目录下已经有了一份同名文件，则输出取决于标志位 bForceOverwrite。该标志位置位时，覆盖目标文件。

表 7-11：createFile

long SIQFileSvc::createFile(const QString& strFilePath, bool bForceOverwrite = false);	
参数	含义
strFilePath	需要创建的文件。
bForceOverwrite	如文件已存在，覆盖该文件。
返回值	创建文件任务的执行状态。

示例：
另见章节 7.2.4 和 7.2.5（“分步示例”）

创建目录(createFolder)

该函数可用于创建一个目录。标志位 bCreateParentFolders 置位时，如果没有父目录，在路径中创建所有父目录。

表 7-12：createFolder

long SIQFileSvc::createFolder(const QString& strFolderPath, bool bCreateParentFolders = false);	
参数	含义
strFolderPath	需要创建的目录。
bCreateParentFolders	一同创建父目录。
返回值	创建目录任务的执行状态。

重命名目录(renameFolder)

该函数可用于重命名一个目录。

表 7-13：renameFolder

long SIQFileSvc::renameFolder(const QString& strFolderPath, const QString& strNewBaseName);	
参数	含义
strFolderPath	要重命名的目录。
strNewBaseName	目录的新名称（不含路径）。
返回值	重命名任务的执行状态。

复制文件(copyFile/copyFileEx)

该函数可复制文件。如果在目标目录下已经有了一份同名文件，则输出取决于标志位 bForceOverwrite。该标志位置位时，覆盖目标文件。

标志位 bForceOverwrite 没有置位时，在异步调用中会发出信号 askOverwrite。客户端因此可以在系统运行时查询覆盖状态，并通过 sendReply 作出响应。

表 7-14：copyFile

long SIQFileSvc::copyFile(	const QString& strSourceFilePath, const QString& strDestPath, bool bForceOverwrite = false);
参数	**含义**
strSourceFilePath	源文件。
strDestPath	目标路径。
bForceOverwrite	如文件已存在，覆盖该文件。
返回值	复制任务的执行状态。

在异步调用中会在调用后返回一个句柄，用于之后标识调用。

该函数发送以下信号：progress, completed，有时也发送 askOverwrite 和 canceled 。

表 7-15：copyFileEx

long SIQFileSvc::copyFileEx(	const QString& strSourceFilePath, const QString& strDestPath, long& nRequestId, bool bForceOverwrite = false);
参数	**含义**
strSourceFilePath	源文件。
strDestPath	目标路径。
nRequestId	用于标识异步调用的句柄(handle)。
bForceOverwrite	如文件已存在，覆盖该文件。
返回值	函数的执行状态。

函数 sendReply 是对信号 askOverwrite 的回复。

表 7-16：sendReply

long SIQFileSvc::sendReply(	long nRequestId, WhatToDoEnum reply);
参数	**含义**
nRequestId	用于标识异步调用的句柄(handle)。
reply	应答是否应该覆盖文件： DO_YES ➔ 覆盖目标文件。 DO_NO ➔ 不覆盖目标文件。
返回值	应答的执行状态。

示例：
参见章节 7.2.4（同步）和 7.2.5（异步）
（“分步示例”）

移动文件(moveFile/moveFileEx)

该函数可移动文件。如果在目标目录下已经有了一份同名文件，则输出取决于标志位 bForceOverwrite。该标志位置位时，覆盖目标文件。

标志位 bForceOverwrite 没有置位时，在异步调用中会发出信号 askOverwrite。客户端因此可以在系统运行时查询覆盖状态。

表 7-17：moveFile

long SIQFileSvc::moveFile(	const QString& strSourceFilePath, const QString& strDestFilePath, bool bForceOverwrite = false);
参数	含义
strSourceFilePath	源文件。
strDestPath	目标路径。
bForceOverwrite	如文件已存在，覆盖该文件。
返回值	移动任务的执行状态。

在异步调用中会在调用后返回一个句柄，用于标识调用。

该函数发送以下信号：progress, completed，有时也发送 askOverwrite 和 canceled 。

表 7-18：moveFileEx

long SIQFileSvc::moveFileEx(	const QString& strSourceFilePath, const QString& strDestFilePath, long& nRequestId, bool bForceOverwrite = false);
参数	含义
strSourceFilePath	源文件。
strDestPath	目标路径。
nRequestId	用于标识异步调用的句柄(handle)。
bForceOverwrite	如文件已存在，覆盖该文件。
返回值	函数的执行状态。

函数 sendReply 是对信号 askOverwrite 的回复。

表 7-19：sendReply

long SIQFileSvc::sendReply(	long nRequestId, WhatToDoEnum reply);
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。
reply	应答是否应该覆盖文件： DO_YES ➔ 覆盖目标文件。 DO_NO ➔ 不覆盖目标文件。
返回值	应答的执行状态。

删除文件/目录(remove/removeEx)

该函数可删除目录或文件。

表 7-20：remove

long SIQFileSvc::remove(const QString& strPath);	
参数	含义
strPath	需要删除的目录或文件。
返回值	删除任务的执行状态。

在异步调用中会在调用后返回一个句柄，用于标识调用。

该函数发送以下信号：progress、completed，有时也发送 canceled。

表 7-21：removeEx

long SIQFileSvc::removeEx(const QString& strPath, long& nRequestId);	
参数	含义
strPath	需要删除的目录或文件。
nRequestId	用于标识异步调用的句柄(handle)。
返回值	函数的执行状态。

示例：
参见章节 7.2.4（同步）和 7.2.5（异步）
（“分步示例”）

检查文件/目录是否存在(exist)

该函数可检查文件或目录是否存在。

表 7-22：exist

long SIQFileSvc::exist(const QString& strPath, bool &bExist);	
参数	含义
strPath	需要删除的目录或文件。
bExist	查询结果。
返回值	查询任务的执行状态。

选择零件程序(select)

该函数可选择一个 NC 被动文件系统中保存的程序，以在特定通道中执行。只有程序为可执行文件时，比如零件程序，该函数才有用。

表 7-23：select

long SIQFileSvc::select(const QString& strProgramPath, unsigned long nChannel);	
参数	含义
strProgramPath	需要选择的文件。
nChannel	执行程序的通道。
返回值	选择任务的执行状态。

如需撤销零件程序的选择，为该文件(strProgramPath)传送“//NC/MPF0”。

执行外部程序(extmod)

该函数可选择一个外部程序（比如：CF 卡或硬盘上的程序）在指定通道中执行。只有程序为可执行文件时，比如零件程序，该函数才有用。

表 7-24：extmod

long extmod(const QString& strProgramPath, unsigned long nChannel, const QString& strNcuName = "", ExtModCallEnum enumCallMode = EXTMOD_START);	
参数	含义
strProgramPath	需要选择的文件。
nChannel	执行程序的通道。
strNcuName	无含义（不传送任何内容）
enumCallMode	无含义（不传送任何内容）
返回值	选择任务的执行状态。

在选择外部程序后，可通过“NC start”直接启动该程序。

查询特性(getAttributes)

该函数可返回文件或目录的特性。

表 7-25：getAttributes

long SIQFileSvc::getAttributes(const QString& strPath, NodeInfo& rAttributes);	
参数	含义
strPath	文件或目录的路径
rAttributes	查询出的特性。
返回值	查询任务的执行状态。

示例：
另见章节 7.2.6“分步示例”

转换路径(getRealPath/getLogicalPath)

该函数可以在逻辑路径和实际路径之间转换。

表 7-26：getRealPath

long SIQFileSvc::getRealPath(const QString& strLogicalPath, QString& strRealPath);	
参数	含义
strLogicalPath	需要转换的逻辑路径。
strRealPath	转换得出的实际路径。
返回值	转换任务的执行状态。

表 7-27：getLogicalPath

long SIQFileSvc::getLogicalPath(const QString& strRealPath, QString& strLogicalPath);	
参数	含义
strRealPath	需要转换的实际路径。
strLogicalPath	转换得出的逻辑路径。
返回值	转换任务的执行状态。

示例：另见章节 7.2.7“分步示例”

设置访问标记(setAccessMask)

该函数设置 NC 文件系统中文件或目录的访问标记。

表 7-28：setAccessMask

long SIQFileSvc::setAccessMask(const QString& strPath, const QString& strNewAccessMask);	
参数	含义
strPath	文件或目录的路径。
strNewAccessMask	需要设置的访问标记。
返回值	设置任务的执行状态。

检查 NC 路径(isPathNC)

该函数可检查传送的路径是 NC 文件系统中的文件还是目录。

表 7-29：isPathNC

long SIQFileSvc::moveFileEx(const QString& strPath, bool& bYesNo);	
参数	含义
strPath	文件或目录的路径。
bYesNo	查询结果。
返回值	查询任务的执行状态。

查询可用硬盘容量(getDiskFreeSpace)

该函数根据逻辑路径返回某个硬盘分区的可用容量和总容量。

表 7-30：getDiskFreeSpace

long SIQFileSvc::getDiskFreeSpace(const QString& strDirectoryName, qint64& nFreeBytesAvailable, qint64& nTotalNumberOfBytes);	
参数	含义
strDirectoryName	路径。
nFreeBytesAvailable	可用硬盘容量。
nTotalNumberOfByte	总硬盘容量。
返回值	查询任务的执行状态。

设置关于 TCU-USB 驱动器的通知

该函数可设置关于 TCU-USB 驱动器和网络驱动器连接和断开的通知。

表 7-31：registerForDriveNotifications

long SIQFileSvc::registerForDriveNotifications(const QStringList& arrDrivePaths);	
参数	含义
arrDrivePaths	设备路径的列表。
返回值	设置任务的执行状态。

取消异步调用(cancelRequest)

该函数可取消异步调用。该函数发送信号“canceled”。

表 7-32：cancelRequest

long SIQFileSvc::cancelRequest(long nRequestID);	
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。
返回值	函数的执行状态。

7.3.4 信号

概述

上文 7.3.3 说明的函数会发送一些信号。这些信号在下文详细说明。

信号“progress”

该信号返回异步调用的执行进度。

表 7-33：progress

void progress(long nRequestId, long nProgressDone);	
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。
nProgressDone	任务的进度，百分比值。

信号“completed”

该信号在异步调用结束但不返回任何结果时发送。

表 7-34：completed

void completed(long nRequestId, long errCode);	
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。
errCode	任务的执行状态。

信号“askOverwrite”

该信号在执行 copyFileEx 函数（复制文件）或者 moveFileEx 函数（移动文件）、发现已存在目标文件时发送。

可利用函数 sendReply 发送回复。该调用的句法在函数 copyFileEx 和 moveFileEx 中详细说明。

表 7-35：askOverwrite

void askOverwrite(	long nRequestId, NodeInfo& rSourceNode, NodeInfo& rDestNode);
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。
rSourceNode	需要删除的文件的特性。
rDestNode	已存在的文件的特性。

信号“listCompleted”

该信号在函数 listFolderEx（显示目录）结束后发送。

表 7-36：listCompleted

void listCompleted(	long nRequestId, NodeInfoArray& rNodes, long errCode);
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。
rNodes	目录的内容。
errCode	任务的执行状态。

信号“canceled”

该信号在函数 cancelRequest（取消异步调用）成功执行后发送。

表 7-37：canceled

void canceled(long nRequestId);	
参数	含义
nRequestId	用于标识异步调用的句柄(handle)。

信号“driveMounted”

该信号在成功安装了 TCU-USB 设备或网络驱动器后发送。

表 7-38：driveMounted

void driveMounted(const QString& strLogicalPath);	
参数	含义
strLogicalPath	驱动器的名称。

信号“driveUnmounted”

该信号在成功卸除了 TCU-USB 设备或网络驱动器后发送。

表 7-39：driveUnmounted

void driveUnmounted(const QString& strLogicalPath);	
参数	含义
strLogicalPath	驱动器的名称。

7.3.5 帮助类 NodeInfo

该类包含了文件或目录的特性。

表 7-40：

```
class NodeInfo
{

public:
...
        QString strPath;
        QString strRealPath;
        bool bFolder;
        QString strBaseName;
        QString strAccessMask;
        long nSize;
        long nLastWriteTimestamp;
        QString wstrDescription;
};
```

参数	含义
strPath	逻辑绝对路径。 （示例："//NC/MPF.DIR/TEST.MPF"）
strRealPath	逻辑路径转换而成的实际路径。 （示例："/_N_MPF_DIR/_N_TEST_MPF"）
bFolder	表明对象是一个文件还是目录。
strBaseName	文件主名称或路径。 （示例："TEST.MPF"）
strAccessMask	文件/目录的存取权限。 （示例： "77777")
nSize	以字节为单位的文件大小。 对于目录，该值为-1。
nLastWriteTimestamp	上一次更改的时间戳“time_t”，即从 1970 年 1 月 1 日晚上 12 点起（UTC 时间）起经过的秒数。
wstrDescription	简明文本标识（只针对 NC 文件系统）

时间的显示应使用以下代码：

```
QDateTime date;
date.setTime_t(node.nLastWriteTimestamp);
QString szDate = date.toString("dd.MM.yy hh:mm");
```

7.4 常见问题(FAQ)

7.4.1 一般常见问题

是在栈上还是在堆上创建 SIQFileSvc 对象？

SIQFileSvc 类的对象只有少数的实例数据，在创建和删除时费时较短。因此，无论是在栈上还是在堆上都可以创建 SIQFileSvc 对象。

此处要建议您删除不再需要使用的 SIQFileSvc 对象，以便再次释放资源。因此对于需要使用 SIQFileSvc 调用的类而言，建议您将此类对象声明为私有成员变量。

可以有多个 SIQFileSvc 对象吗？

在一个类内可以创建多个 SIQFileSvc 对象。但是我们还是建议您作为私有成员变量创建一个 SIQFileSvc 对象，将该对象用于调用。

需要调用 disconnect 命令断开信号与槽的关联吗？

断开信号与槽之间的关联是由 Qt 在后台执行的。您无需显式编程 disconnect。

每个异步调用都需要一个 SIQFileSvc 对象吗？

不是。一个 SIQFileSvc 对象完全是一个用户接口，它可以启动多个不同任务。也就是说您可以通过一个 SIQFileSvc 对象启动多个异步调用。

但是每个异步调用都需要一个所谓的“RequestId”，它是一个“long”型变量。在启动一个此类调用后，“RequestId”包含一个唯一值，该值可用于之后标识槽中的各个任务。

没有找到函数“Activate”和“Passivate”。如何装载和卸载零件程序？

相关信息见章节 7.1.2 下的段落“Powerline 升级版的说明”。

8 TRACE 服务

8

本章主要内容

本章主要介绍 SINUMERIK Operate 编程包
中服务组件下的 TRACE 服务。TRACE 服务提供相关函数记录 NC/PLC 变量与信号的变化。

注

详细的说明参见随 SINUMERIK Operate 编程包发货的引用
“Referenz_SITrace.pdf”。

8.1 分步示例

概述

以下章节将分步介绍 TRACE 服务的使用方式。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例程序。

表 8-1：示例一览

应用	示例	章节
以 XML 格式记录 NC 数据	slextracesvr	8.1.2

此处只举例说明或描述与 TRACE 服务相关的内容。源文件的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

开发环境

为 TRACE 服务示例程序建立开发环境所需的步骤和 5.3 章指出的步骤完全一致。

! **重要提示**

为节省资源，目标系统上 TRACE 服务没有激活。如需激活 TRACE 服务，必须在文件“/oem/sinumerik/him/cfg/systemconfiguration.ini”中添加以下条目：

```
[services]
SVC012= name:=SlTraceSvc, implementation:=sltracesvc.SlTraceSvc,
process:=SlHmiHost1
```

8.1.1 准备

概述

满足以下前提后，才能从自定义的项目或类中调用 TRACE 服务。

通过 SinuComNC 确定 XML 设置

数据记录的设置需要使用软件 SinuComNC 确定。按如下步骤：

1) 启动 SinuComNC
2) 为会话配置所需的参数
3) 保存会话（比如：保存为“sample.xml”）

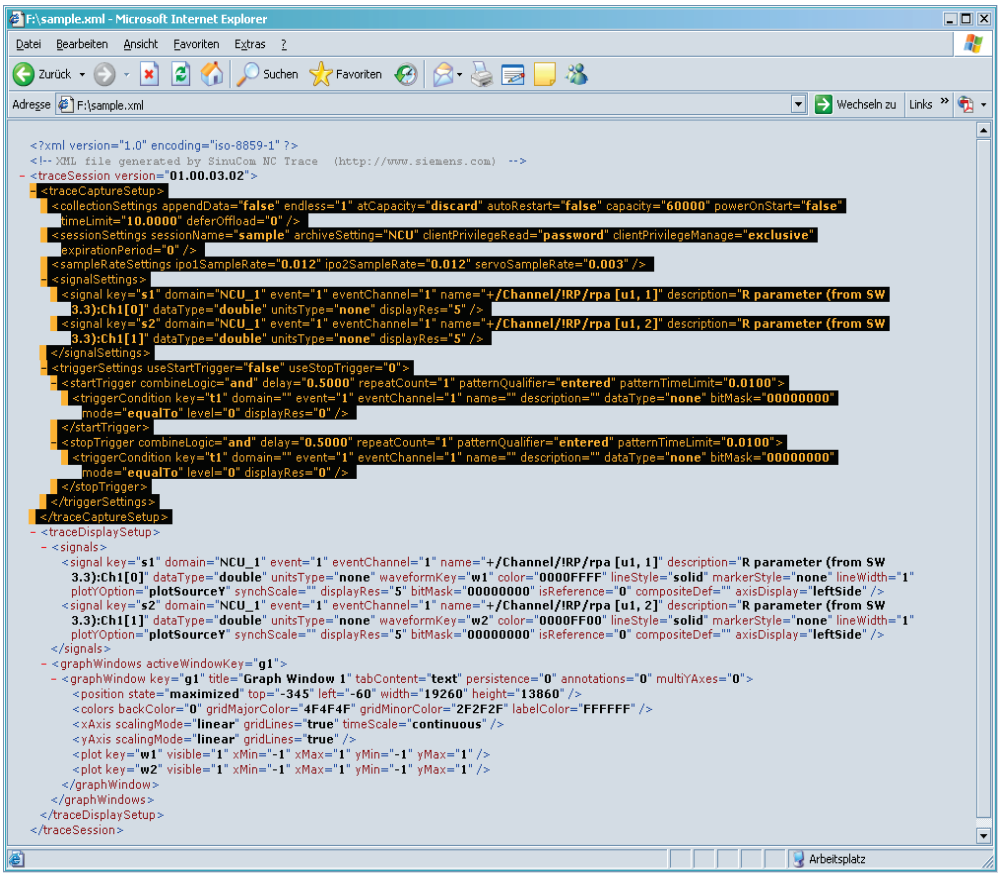


图 8-1:SinuComNC 中的会话示例

保留会话文件中选中的部分，也就是段落“traceCaptureSetup”，删除其他部分。此外还要删除 XML 标题。之后文件内容为：

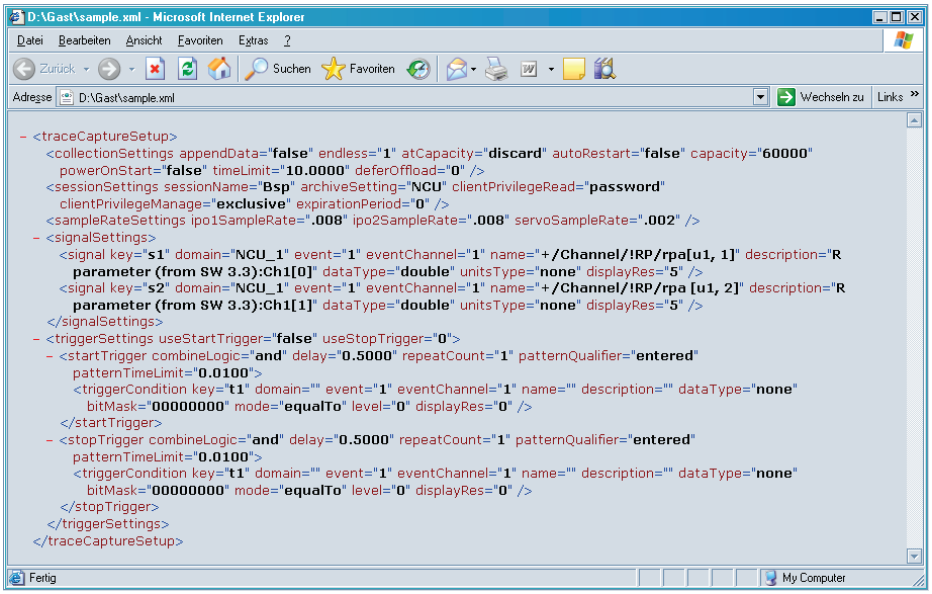


图 8-2:用于 TRACE 服务的 XML 设置

示例中已经插入了一份此类 XML 设置文件(sample.xml)，该文件以插补周期记录 R 参数：R1、R2 和 R3。

注

注意以下几点：

1. XML 设置中的属性“ipo1SampleRate”和“ipo2SampleRate”必须为控制系统插补周期的整数倍值。
2. XML 设置中的属性“servoSampleRate”必须为控制系统伺服周期的整数倍值。

检查库

在项目设置中检查库 `sltraceadapter.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `sltraceadapter.lib`

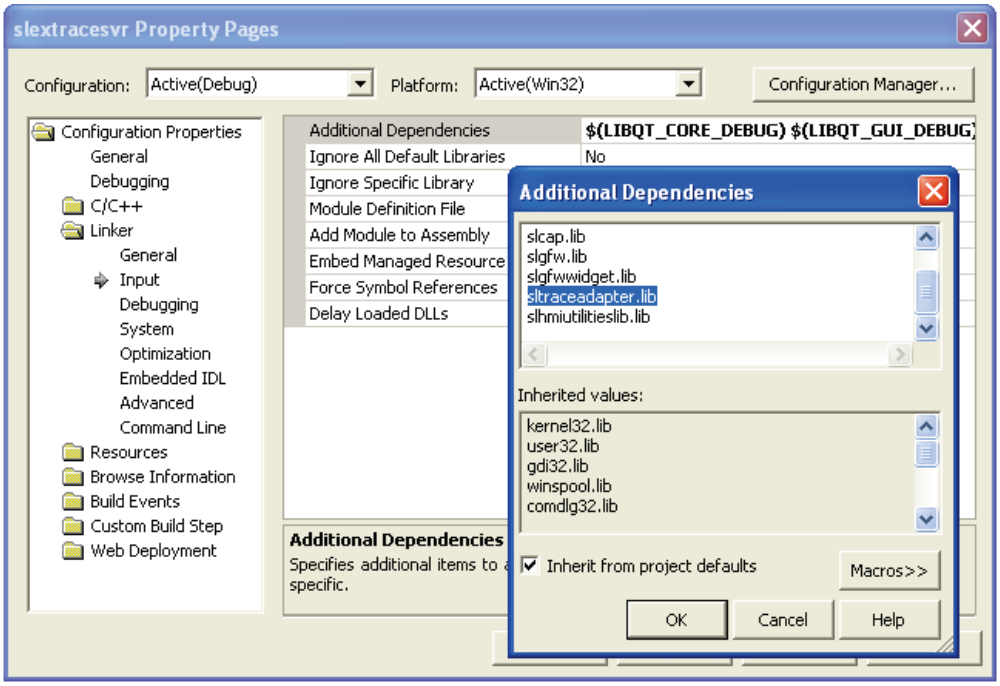


图 8-3:库“sltraceadapter.lib”

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lsltraceadapter
```

头文件

要使用 TRACE 服务的类需要加入头文件“sltrace.h”，配置条目为：

```
#include "sltrace.h"
```

创建 SITraceAdapterPtr 对象

访问 TRACE 服务的接口需要使用 SITraceAdapterPtr 类的对象。这些 Smartpointer 对象可作为私有的成员变量创建：

```
SlTraceAdapterPtr m_pSLTraceAdapter;
```

创建 SITraceQSessConn 指针

访问会话的的接口方法需要使用指向 SITraceQSessConn 对象的指针。该指针可作为私有的成员变量创建：

```
SlTraceQSessConn* m_pISessConn;
```

8.1.2 以 XML 格式记录 NC 数据

概述

下面的例子分步展示了如何启动选中 NC 数据的记录，如何将生成的 XML 格式的记录加入 HMI 中。SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExTraceSvr”。

在本例中，每个步骤是由按下/松开对应的软键控制的，因此在调用该步骤的函数时，始终只有一个软键可被操作。

记录会话的当前状态显示在示例程序屏幕的上半部分。 需要记录的数据和信号在文件“sample.xml”中以对应格式确定。该文件的内容以及数据的记录结果在按下对应软键后显示在屏幕中央。

状态栏显示调用成功。

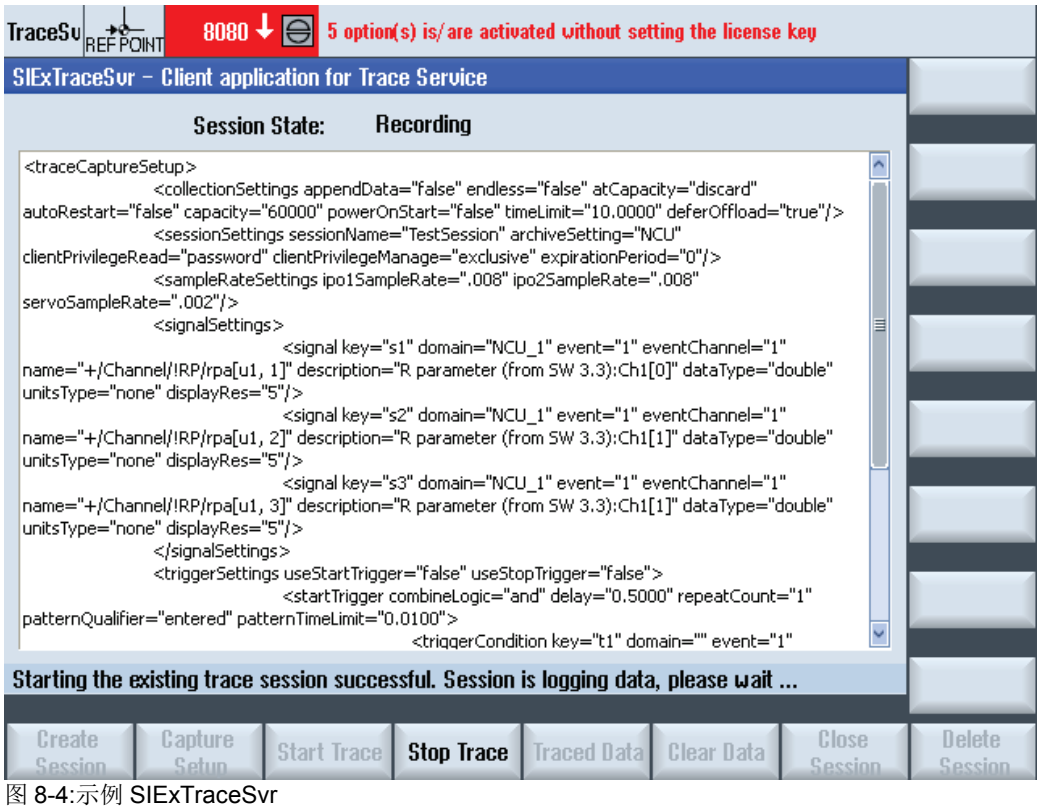


图 8-4:示例 SIExTraceSvr

务必要满足章节 8.1.1 列出的前提条件。

第 1 步

在 TRACE 服务中输入会话名称和口令，
创建记录会话。之后会返回指针 `m_pISessConn`。通过该指针可访问会话。

```
eSlError = m pSLTraceAdapter->CreateSession (szClientName,
                                             szPassword,

                                             szSessionName,
                                             m_pISessConn);
```

第 2 步

现在将成员函数（槽）`OnStateChange` 与 TRACE 服务的信号 `StateChange` 关联在一起。

```
QObject::connect(m pISessionConn,
                 SIGNAL(StateChange(TraceSessStateEnum, SITraceErrEnum)),
                 this,
                 SLOT(OnStateChange(TraceSessStateEnum, SITraceErrEnum));
```

现在将成员函数（槽）`OnDataReady` 与 TRACE 服务的信号 `DataReady` 关联在一起。

```
QObject::connect(m_pISessConn,        SIGNAL(DataReady(ulong)),
                            this,        SLOT(OnDataReady(ulong)));
```

第 3 步

在成员函数 OnStateChange 中可以实现一些需要在会话状态改变时执行的命令。

```
void SlExTraceSvrForm::OnStateChange(TraceSessStateEnum SessionState,
                                                    SlTraceErrEnum Error)
{
        // ToDo:handle session state has changed
}
```

在成员函数 OnDataReady 中可以实现一些在提供由 TRACE 服务记录的数据时执行的命令。

```
void SlExTraceSvrForm::OnDataReady(ulong)
{
        // ToDo:handle trace data is provided
}
```

第 4 步

在一个字符串中读取文件“sample.xml”的内容（含记录用参数）。

```
QString szCaptureSetupFile = "/card/oem/sinumerik/hmi/appl/sample.xml"
QFile qFileXML(szCaptureSetupFile);
if ( qFileXML.open(QFile::ReadOnly | QFile::Text) )
{
        QTextStream    qTextFileXML(&qFileXML);
        szCaptureSetupXML = qTextFileXML.readAll();

       qFileXML.close();
}
```

第 5 步

将 QStringszCaptureSetupXML 中的记录参数赋给会话。

```
eSlError = m_pISessConn->PutCaptureSetup(szCaptureSetupXML, lErrOffset);
```

第 6 步

启动会话。

```
eSlError = m_pISessConn->Start();
```

第 7 步

上一步成功执行并启动会话后，可以显式暂停会话。 示例“sample.xml”最迟在 10 秒后自行结束。只能在状态 traceStateRecording 或 traceStateArmed 中暂停会话。因此首先要查看会话的当前状态。

```
eSlError = m pISessConn->GetSessionState(CurrentSessionState,
                                         eSessionFault);

if ( (CurrentSessionState == traceStateRecording) ||
     (CurrentSessionState == traceStateArmed) )
       eSlError = m_pISessConn->Stop();
```

第 8 步

满足以下某个前提后可以读取数据包：

a） 调用了槽函数 OnDataReady。

b） 显式查询是否可读取记录结果的(DataPresent)返回 true。

此处可在函数 GetDataXml 中利用参数 lGetDataOptions 确定第一个数据包和后续数据包中数据的类型。

```
eSlError = m_pISessConn->DataPresent(fDataReadyToRead);
if ( fDataReadyToRead == true )
{
        if ( !m szSaveDataXML.size() )
            // if first data packet
               lGetDataOptions = traceGetDataFrames +
                                 traceGetDataHeader +
                                 traceGetDataSignals;
        else
               // if subesquent data packets
               lGetDataOptions = traceGetDataFrames;

       eSlError = m_pISessConn->GetDataXml(lGetDataOptions, lMaxDataSize,
                                           szLogDataInXML, fMoreDataLogged);
}
```

这一步可以重复执行，直到 TRACE 服务不再报告有数据可读取（参数 fMoreDataLogged==false）。

第 9 步

现在可以利用以下函数删除 TRACE
服务缓冲器中记录的数据。

```
eSlError = m_pISessConn->ClearData();
```

第 10 步

不需要再次启动当前会话的数据记录时，可以再次断开第 2 步中信号与槽之间的关联。

```
QObject::disconnect(m_pISessConn, SIGNAL(DataReady(ulong)),
                    this,        SLOT(OnDataReady(ulong)));

QObject::disconnect(m pISessConn, SIGNAL(StateChange(TraceSessStateEnum,
                                                      SlTraceErrEnum)),
     this,                         SLOT(OnStateChange(TraceSessStateEnum,
                                                     SlTraceErrEnum)));
```

第 11 步

然后可以用以下函数结束会话：

```
eSlError = m_pISessConn->Close();
```

该步结束后仍处于 TRACE 服务中的会话可用以下方法再次启动，以便继续使用：

```
        eSlError = m_pSLTraceAdapter->OpenSession (szClientName,
                                                    szPassword,
                                                    szSessionName,
                                                    m_pISessConn);
```

第 12 步

最后将第 1 步中创建的会话从
TRACE 服务的管理中删除。

```
eSlError = m pSLTraceAdapter->DeleteSession (szClientName,
                                             szPassword,
                                             szSessionName);
```

9

9 操作记录服务

本章主要内容

本章主要介绍 SINUMERIK Operate 编程包中隶属于服务组件的操作记录服务。借助操作记录服务您可以访问 SINUMERIK Operate 的操作记录。

另外您还可以在操作记录中添加自定义的条目，以便在自定义条目和受操作记录监控的事件之间建立时间上的关联。比如：可将用户 ID 加入到操作记录中，以便查看某个用户的操作。

9.1 引言

9.1.1 类模型

概述

操作记录可以记录各种操作步骤，以备日后查看。它可以监控各种事件：

- 报警状态变化
- 按键操作
- 通道状态变化
- 窗口切换
- 写入 NCK/PLC 数据
- 文件访问
- 功能调用
- 当前的程序状态

操作记录服务的类模型主要由以下类组成：

- SIQTrp

SIQTrp 类

借助 SIQTrp 对象您可以读出当前操作记录。另外您还可以在其中添加自定义的条目。

读取操作记录这一任务既有异步调用方式也有同步调用方式。其中，同步调用会阻塞调用所在的线程，直到任务完成。而在异步调用中，SIQTrp 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

注

SIQTrp 对象只能应用在 Qt 主线程中。

9.1.2 术语解释

同步调用

同步调用在任务执行完后才会返回，即
当前线程会被一直阻塞。这会干扰
事件处理，例如在同步调用时
阻止输入和显示。因此比较耗时的
调用应进行异步调用。

异步调用

只要任务被发送给操作记录服务，
异步调用就返回。这尤其意味着，所提供的故障代码不代表任务是否成功完成，而只是任务被成功发送的回复。例如当未正确提供调用参数时，则会出错。真正的任务状态在回调操作记录服务时提供（信号与槽机制）。

记录类型

表 9-1：记录类型

记录类型	描述
当前记录	该记录包含了当前条目。它设计为环形缓冲器，旧条目被自动覆盖。
Crash 日志	出现严重错误时，操作记录会将当前日志的内容保存到所谓的 Crash 日志中。这些备份条目因此不会再被新条目覆盖。只有在下一次发生重大错误后才会覆盖 Crash 日志。

注

有两种记录格式。 在记录函数(getLog, getLogAsync, getCrashLog, getCrashLogAsync)中没有设置可选参数“lOptions”时，记录格式为 SINUMERIK Operate V2.6 中的格式，以保证兼容性。设置“SLTRP_GETLOG_OPTION_NEWFORMAT”后，记录格式为经过优化的格式。

9.2 分步示例

概述

以下章节将分步介绍操作记录服务的使用方式。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例。

表 9-2：示例一览

应用	示例	章节
同步读取操作记录	slexsynctrp	9.2.2
异步读取操作记录	slexasynctrp	9.2.3

此处只举例说明或描述与操作记录服务相关的内容。源文件的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

开发环境

为示例程序建立开发环境所需的步骤和 5.3 章指出的步骤完全一致。

9.2.1 准备

概述

完成以下准备工作后，才能从自定义的项目或类中调用操作记录服务。

检查库

在项目设置中检查库 `sltrp.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `sltrp.lib`

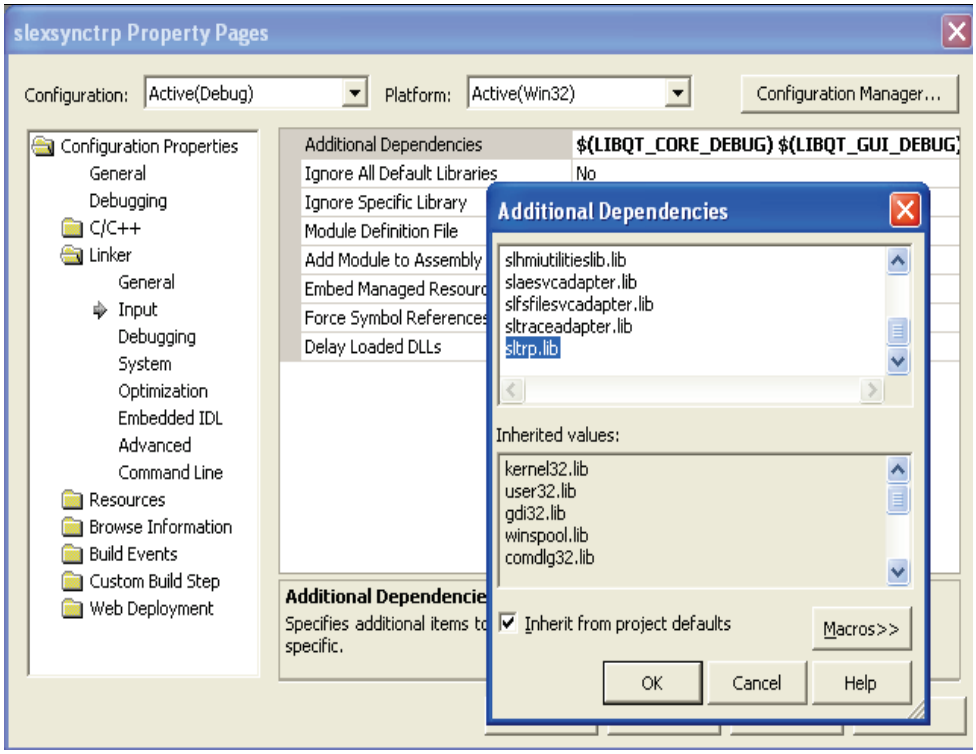


图 8-1:库“sltrp.lib”

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lsltrp
```

头文件

要使用操作记录服务的类需要加入头文件 `sltrpqsvc.h` 和 `sltrpids.h`，配置条目为：

```
#include "sltrpqsvc.h"
#include "sltrpids.h"
```

创建 SIQTrp 对象

访问操作记录服务的接口需要使用 SIQTrp 类的一个对象。该对象可作为私有的成员变量创建：

```
SlQTrp m_trpServer;
```

9.2.2 同步读取操作记录

概述

下面的示例分步介绍了如何同步读取操作记录并在其中加入自定义的条目。
SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExSyncTrp”。

在该示例程序中，在初始化操作记录(init)、激活操作记录(activate)并注册用户后可以通过“make user entry”在操作记录中加入自定义条目。 通过“get log”可以读出当前操作记录。状态栏显示调用成功。

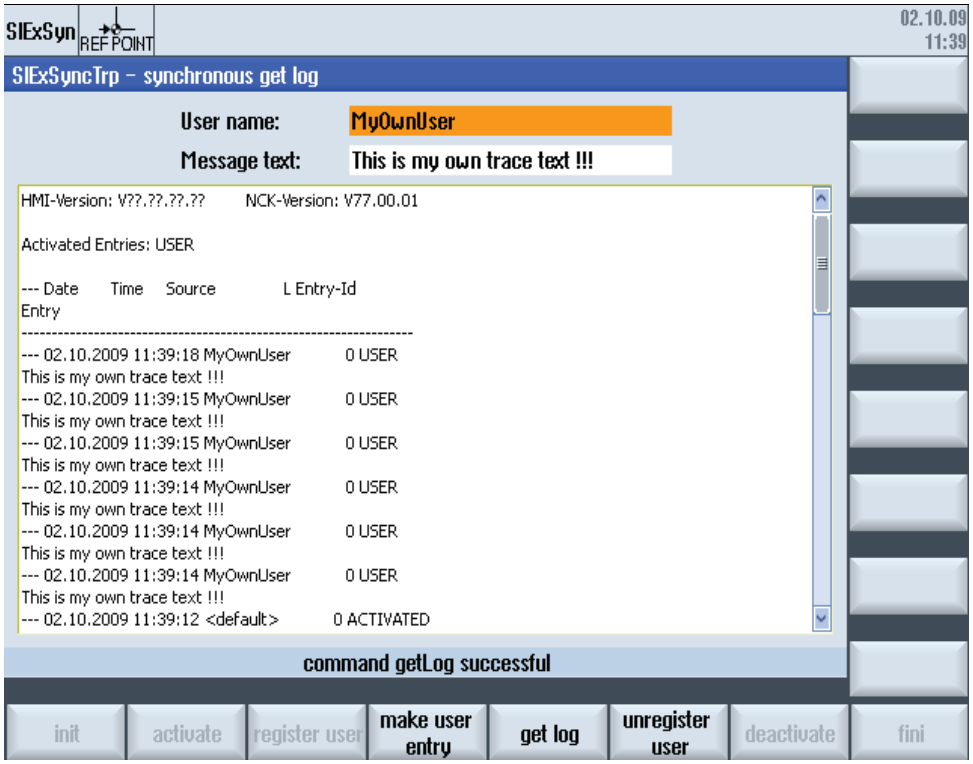


图 8-2:示例 SIExSyncTrp

务必要满足章节 9.2.1“准备”列出的前提条件。

第 1 步

初始化操作记录对象(SITrp)。返回值 0 表示初始化成功。

```
if ( 0 == (lRetVal = m trpServer.init()) )
{
   szRetVal = SUCCESS INIT;
   ...
}
```

第 2 步

激活操作记录。在本例中只激活 User 条目。

```
unsigned char ucaTripIds[2];
ucaTripIds[0] = TRIP_USER;
ucaTripIds[1] = TRIP_INVALID;

if ( 0 == (lRetVal = m_trpServer.activateTrip(ucaTripIds)) )
{
   szRetVal = SUCCESS ACTIVATE;
   ...
}
```

第 3 步

注册用户，以便在操作记录中加入自定义条目。用户名之后会显示在操作记录中。返回的参数 m_lUserId 用于标识发送方。

```
QString szUserName = "MyOwnUser";
if ( 0 == (lRetVal = m_trpServer.registerUser(szUserName, m_lUserId)) )
{
   szRetVal = SUCCESS REGISTER USER;
   ...
}
```

第 4 步

在操作记录中加入自定义条目。其中，参数 m_lUserId 来自调用 registerUser。

```
QString szUserEntry = "This is my own trace text !!!";
m_trpServer.traceUserString(m_lUserId, 0, szUserEntry);
```

第 5 步

将当前操作记录写入到一份文件中。在本例中，记录文件的内容直接载入 QTextBrowser 小部件中。

```
QString szFile = "/card/oem/sinumerik/hmi/appl/action.log";
if ( 0 == (lRetVal = m_trpServer.getLog(szFile,
                                        SLTRP_GETLOG_OPTION_NEWFORMAT)) )
{
   szRetVal = SUCCESS GETLOG;
   ...

   // show log in textbrowser
   QFile qFile(szFile);
   if (qFile.open(QFile::ReadOnly | QFile::Text))
   {
           // read input of the file
           QTextStream qTextStream(&qFile);
           m_pTbDisplayLog->setText( qTextStream.readAll() );

           // close file
           qFile.close();
   }
}
```

注

取决于当前操作记录的大小，同步函数“getLog”可能需要花费很长时间才能返回，可能会阻塞整个 SINUMERIK Operate 。 备选方法是使用异步函数“getLogAsync”。

第 6 步

注销注册的用户。其中，参数 m_lUserId 来自调用 registerUser。

```
if ( 0 == (lRetVal = m trpServer.unregisterUser(m lUserId)) )
{
   szRetVal = SUCCESS UNREGISTER USER;
   ...
}
```

第 7 步

关闭操作记录。

```
if ( 0 == (lRetVal = m trpServer.deactivateTrip()) )
{
   szRetVal = SUCCESS_DEACTIVATE;
   ...
}
```

第 8 步

终止操作记录对象(SITrp)。

```
if ( 0 == (lRetVal = m trpServer.fini()) )
{
   szRetVal = SUCCESS FINI;
   ...
}
```

9.2.3 异步读取操作记录

概述

下面的示例分步介绍了如何异步读取操作记录并在其中加入自定义的条目。
SINUMERIK Operate 编程包中有一个展示如何完成该任务的可执行示例程序：项目“SIExAsyncTrp”，该示例程序
的结构和 9.2.2 章描述的同步读取操作记录的示例相同。

务必要满足章节 9.2.1“准备”列出的前提条件。

第 1 步

任务的执行结果通过信号通知操作记录服务。该信号由以下函数（槽）接收。该函数的定义在第 7 步(slotAsyncFinished)中变得清晰明确。

```
private slots:
   void slotAsyncFinished (long lAsyncHandle, SlTrpErrorEnum eError);
```

第 2 步

现在将成员函数（槽）与操作记录服务的信号 `onAsyncFinished` 关联在一起。

```
QObject::connect(  &m_trpServer,
                   SIGNAL(onAsyncFinished (long, SlTrpErrorEnum)),
                   this,
                   SLOT(slotAsyncFinished (long, SlTrpErrorEnum)));
```

第 3 步

初始化操作记录对象(SITrp)。返回值 0 表示初始化成功。

```
if ( 0 == (lRetVal = m trpServer.init()) )
{
   szRetVal = SUCCESS INIT;
   ...
}
```

第 4 步

激活操作记录。在本例中只激活 User 条目。

```
unsigned char ucaTripIds[2];
ucaTripIds[0] = TRIP_USER;
ucaTripIds[1] = TRIP_INVALID;
```

```
if ( 0 == (lRetVal = m_trpServer.activateTrip(ucaTripIds)) )
{
   szRetVal = SUCCESS_ACTIVATE;
   ...
}
```

第 5 步

注册用户，以便在操作记录中加入自定义条目。用户名可随意选择，之后会显示在操作记录中。返回的参数 `m_lUserId` 用于标识发送方。

```
QString szUserName = "MyOwnUser";
if ( 0 == (lRetVal = m_trpServer.registerUser(szUserName, m_lUserId)) )
{
   szRetVal = SUCCESS_REGISTER_USER;
   ...
}
```

第 6 步

在操作记录中加入自定义条目。其中，参数 `m_lUserId` 来自调用 `registerUser`。

```
QString szUserEntry = "This is my own trace text !!!";
m_trpServer.traceUserString(m_lUserId, 0, szUserEntry);
```

第 7 步

向操作记录服务传送任务：将 ASCII 格式的操作记录写入指定文件中。返回的句柄 `m_lAsyncHandle` 用于标识任务。

```
QString szFile = "/card/oem/sinumerik/hmi/appl/action.log";
if ( 0 == (lRetVal = m_trpServer.getLogAsync(szFile,
                         SLTRP_GETLOG_OPTION_NEWFORMAT, m_lAsyncHandle)) )
{
   ...
}
```

第 8 步

调用立即返回。调用了槽 slotAsyncFinished 后，当前操作记录才返回。在本例中，记录文件的内容直接载入 QTextBrowser 小部件中。

```
void SlExAsyncTrpForm::slotAsyncFinished (long lAsyncHandle,
                                          SlTrpErrorEnum eError)
{
   if ( lAsyncHandle == m lAsyncHandle )
   {
           ...

           // show log in textbrowser
           QFile qFile(szFile);
   if (qFile.open(QFile::ReadOnly | QFile::Text))
           {
                   // read input of the file
                   QTextStream qTextStream(&qFile);
                   m_pTbDisplayLog->setText( qTextStream.readAll() );

                   // close file
                   qFile.close();
           }
           ...
   }
}
```

第 9 步

注销注册的用户。其中，参数 m_lUserId 来自调用 registerUser。

```
if ( 0 == (lRetVal = m_trpServer.unregisterUser(m_lUserId)) )
{
   szRetVal = SUCCESS UNREGISTER USER;
   ...
}
```

第 10 步

关闭操作记录。

```
if ( 0 == (lRetVal = m trpServer.deactivateTrip()) )
{
   szRetVal = SUCCESS_DEACTIVATE;
   ...
}
```

第 11 步

终止操作记录对象(SITrp)。

```
if ( 0 == (lRetVal = m_trpServer.fini()) )
{
   szRetVal = SUCCESS FINI;
   ...
}
```

9.3 SIQTrp 引用

9.3.1 定义

概述

借助 SIQTrp 对象您可以读出当前操作记录。另外您还可以在其中添加自定义的条目。

读取操作记录这一任务既有异步调用方式也有同步调用方式。其中，同步调用会阻塞调用所在的线程，直到任务完成。而在异步调用中，SIQTrp 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

注

为提高性能，我们建议您创建一个全局 SIQTrp 对象，而不是每次在需要时才初始化 SIQTrp 对象。

9.3.2 初始化和终止函数

初始化(init)

初始化 SIQTrp 对象。该函数作为第一个函数调用。

表 9-3：init

long SIQTrp::init(int argc = 0, const char* pArgv[] = 0);	
参数	含义
argc	无含义（不传送任何内容）
pArgv	无含义（不传送任何内容）
返回值	函数的执行状态

终止(fini)

释放 SIQTrp 对象占用的资源。该函数作为最后一个函数调用。

表 9-4：fini

long SIQTrp::fini();	
参数	含义
返回值	函数的执行状态

9.3.3 在记录中加入自定义条目

registerUser

注册操作记录用户 `pszUserName`。返回的 `lUserID` 用于标识操作记录用户。在操作记录中，`pszUserName` 显示为该条目的来源。

表 9-5：registerUser

SITrpErrorEnum SIQTrp::registerUser(const QString& pszUserName, long& lUserId);	
参数	含义
pszUserName	操作记录中的用户名
lUserId	用户 ID（用于输出 TRACE）
返回值	函数的执行状态

unregisterUser

注销用 `registerUser` 注册的操作记录用户。`lUserId` 是由 `registerUser` 返回的用户 ID。

表 9-6：unregisterUser

SITrpErrorEnum SIQTrp::unregisterUser(long& lUserId);	
参数	含义
lUserId	用户 ID
返回值	函数的执行状态

suspendUser

暂停用 `registerUser` 注册的操作记录用户。`lUserId` 是由 `registerUser` 返回的用户 ID。

表 9-7：suspendUser

SITrpErrorEnum SIQTrp::suspendUser(long lUserId);	
参数	含义
lUserId	用户 ID
返回值	函数的执行状态

resumeUser

恢复之前用 `suspendUser` 暂停的用户。

表 9-8：resume

SITrpErrorEnum SIQTrp::resumeUser(long lUserId);	
参数	含义
lUserId	用户 ID
返回值	函数的执行状态

traceUserString

在操作记录中加入一条用户条目 `lUserId`。但只有在激活操作记录(`activateTrip`)时一同设置了对应 ID(`TRIP_USER`)时，才能够加入该条目。

表 9-9：traceUserString

SITrpErrorEnum SIQTrp::traceUserString(long lUserId, long lTraceLevel, const QString& rszMsg);	
参数	含义
lUserId	用户 ID
lTraceLevel	无含义（传送 0）
rszMsg	文本的条目。
返回值	函数的执行状态

9.3.4 读取记录

getLog

创建一份包含当前操作记录的 ASCII 文件。该函数同步调用，会阻塞 SINUMERIK Operate ，直到文件成功创建。

表 9-10：getLog

SITrpErrorEnum SIQTrp::getLog(const QString& pszFileName, long lOptions);	
参数	含义
pszFileName	文件名/输出文件的路径。
lOptions	记录的格式（可选）。 (SLTRP_GETLOG_OPTION_NEWFORMAT ➔经过优化的新格式）
返回值	函数的执行状态

getLogAsync

创建一份包含当前操作记录的 ASCII 文件。函数异步调用，立即返回。该函数结束后会发送信号 `onAsyncFinish`。信号 `onProgressInfo` 可指明执行进度。

表 9-11：getLogAsync

SITrpErrorEnum SIQTrp::getLogAsync(const QString& pszFileName, long lOptions, long& rHandle);	
参数	含义
pszFileName	文件名/输出文件的路径。
lOptions	记录的格式（可选）。 (SLTRP_GETLOG_OPTION_NEWFORMAT ➔经过优化的新格式）
rHandle	用于标识任务的句柄。
返回值	函数的执行状态

getCrashLog

创建一份包含当前 Crash 操作记录的 ASCII 文件。该函数同步调用，会阻塞 SINUMERIK Operate ，直到文件成功创建。

表 9-12：getCrashLog

SITrpErrorEnum SIQTrp::getCrashLog(const QString& pszFileName, long lOptions);	
参数	含义
pszFileName	文件名/输出文件的路径。
lOptions	记录的格式（可选）。 (SLTRP_GETLOG_OPTION_NEWFORMAT ➔经过优化的新格式）
返回值	函数的执行状态

getCrashLogAsync

创建一份包含当前 Crash 操作记录的 ASCII 文件。函数异步调用，立即返回。该函数结束后会发送信号 `onAsyncFinish`。信号 `onProgressInfo` 可指明执行进度。

表 9-13：getCrashLogAsync

SITrpErrorEnum SIQTrp::getCrashLogAsync(const QString& pszFileName, long lOptions, long& rHandle);	
参数	含义
pszFileName	文件名/输出文件的路径。
lOptions	记录的格式（可选）。 (SLTRP_GETLOG_OPTION_NEWFORMAT ➔经过优化的新格式）
rHandle	用于标识任务的句柄。
返回值	函数的执行状态

cancelGetLog

取消已经启动的异步函数（`getLogAsync` 或 `getCrashLogAsync`）。

表 9-14：cancelGetLog

SITrpErrorEnum SIQTrp::cancelGetLog(long lHandle);	
参数	含义
rHandle	用于标识任务的句柄。
返回值	函数的执行状态

9.3.5 管理记录

activateTrip

激活操作记录。此时可通过 pucTripIds 指定将哪个 TRACE 事件加入到操作记录中。TACE ID 应该用 TRIP_INVALID 结束，比如：

```
unsigned char ucaTripIds[3];
ucaTripIds[0] = TRIP_HMI_START;
ucaTripIds[1] = TRIP_KEY_PRESSED;
ucaTripIds[2] = TRIP_INVALID;
```

表 9-15：activateTrip

SlTrpErrorEnum SlQTrp::activateTrip(const unsigned char *pucTripIds);	
参数	**含义**
pucTripIds	TRACE ID （另见章节 9.3.7 “枚举数和定义”下的段落 “pucTripIds”）
返回值	函数的执行状态

deactivateTrip

关闭操作记录。

表 9-16：deactivateTrip

SlTrpErrorEnum SlQTrp::deactivateTrip(void);	
参数	**含义**
返回值	函数的执行状态

getTripIds

查询当前激活并加入到操作日志中的 TRACE 事件。

表 9-17：getTripIds

SlTrpErrorEnum SlQTrp::getTripIds(unsigned char *pucTripIds, long lMaxTripIds);	
参数	**含义**
pucTripIds	TRACE ID （另见章节 9.3.7 “枚举数和定义”下的段落 “pucTripIds”）
lMaxTripIds	需要查询的 TRACE 事件的最大数量。
返回值	函数的执行状态

示例：

```
unsigned char* ucaTripIds = 0;
ucaTripIds = new unsigned char[20];

SlTrpErrorEnum eRetVal = m_trpServer.getTripIds(ucaTripIds, 20);

int nIndex = 0;
while (ucaTripIds[nIndex++] != TRIP_INVALID)
{
   // to do
}
```

9.3.6 通知/信号

概述

上文 9.3 说明的函数会发送一些信号。这些信号在下文详细说明。

onAsyncFinished

该信号报告某个异步任务已结束。任务由一同返回的 `lAsyncHandl` 标识。

表 9-18：onAsyncFinished

void onAsyncFinished (long lAsyncHandle, SlTrpErrorEnum Error);	
参数	含义
lAsyncHandle	用于标识任务的句柄。
Error	函数的执行状态

onProgressInfo

该信号返回异步调用的执行进度。任务由一同返回的 `lAsyncHandle` 标识。

表 9-19：onProgressInfo

void onProgressInfo (long lAsyncHandle, float fProgress);	
参数	含义
lAsyncHandle	用于标识任务的句柄。
fProgress	任务的执行进度（0.0 到 1.0）

onStatusChanged

该信号在操作记录被激活或关闭时发送。

表 9-20：onStatusChanged

void onStatusChanged (bool bTripActive);	
参数	含义
bTripActive	在操作记录被激活时返回“true”，被关闭时返回“false”

9.3.7 枚举数和定义

pucTripIds

可记录下列 TRACE 事件。这些事件可以用于激活操作记录（activateTrip）：

表 9-21：pucTripIds

TripIDs	含义
TRIP_HMI_START	记录 HMI 开机过程
TRIP_HMI_EXIT	记录 HMI 关机过程
TRIP_PLC_CRASH	记录 PLC Crash（DB19.DBX0.6 置位会生成一条指出当前程序状态的条目，并写入到 Crash 记录中）
TRIP_KEY_PRESSED	记录按键按下操作。
TRIP_KEY_RELEASED	记录按键松开操作。
TRIP_ALARM	记录报警事件。
TRIP_ALARM_QUIT	记录报警消失事件（仅报警号）。
TRIP_OPEN_WINDOW	记录窗口打开操作。
TRIP_CH_STATE_CHANGED	记录通道状态（程序正在运行）。
TRIP_TOOL_CHANGED	记录换刀操作。
TRIP_OVERRIDE	记录倍率变更操作。
TRIP_WRITE_VAR	记录 NCK/PLC 变量的写操作。
TRIP_NC_CONNECTION	记录 NC 连接的状态。
TRIP_OPMODE_CHANGED2	记录运行方式的切换操作，含机床功能
TRIP_ALARM_QUIT2	记录报警消失事件（完成的报警条目）。
TRIP_PI_CMD_SL	记录 PI 服务。
TRIP_DOM_CMD	记录到 NC 的文件传送操作。
TRIP_TOOL_CHANGED_NAME	记录换刀操作，以 HMI-sl 格式
TRIP_OPEN_WINDOW_AREA	记录窗口切换操作，含操作区域标识
TRIP_USER	记录用户条目。
TRIP_INVALID	结束标识。

10

10 存档服务

本章主要内容

本章主要介绍 SINUMERIK Operate 编程包
中隶属于服务组件的存档服务。存档服务可实现对不同类型存档的创建和读入。支持以下类型：

- 批量调试存档
- 用户存档
- PLC 升级存档
- 穿孔带存档

此外还可对存档服务的函数进行监控。 例如，可有目的地对批量调试（标准操作区域“调试”）作出反应。

10.1 引言

10.1.1 类模型

概述

存档服务的类模型主要由以下的类组成：

- SlArchiveQSvc
- SlArchiveSvcNotifier

SlArchiveQSvc 类

使用 SlArchiveQSvc 对象可以生成和读入不同的存档类型。

创建分为两步：
a) 首先创建一张列出所有需要加入到存档中的文件/目录和任务的列表。
b) 接着根据该列表创建存档。

在读入时存档的内容会解包到相应的路径下。

列表创建（步骤 a）这一任务既可以同步调用也可以异步调用。存档的生成和读取因耗时较长而只能进行异步调用。

同步调用此时会阻塞调用所在的线程，
直到任务完成。而在异步调用中，
SlArchiveQSvc 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

注

SlArchiveQSvc 对象只能应用在 Qt 主线程中。

SlArchiveSvcNotifier 类

使用 SlArchiveSvcNotifier 对象可监控存档服务中正在执行的异步函数从而作出所需响应，比如对标准操作区域“调试”中进行的批量调试作出响应。

10.1.2 术语解释

同步调用

同步调用在任务执行完后才会返回，即当前线程会被一直阻塞。这会干扰事件处理，例如在同步调用时阻止输入和显示。因此比较耗时的调用应进行异步调用。

异步调用

只要任务被发送给存档服务，异步调用就返回。这尤其意味着，所提供的故障代码不代表任务是否成功完成，而只是任务被成功发送的回复。例如当未正确提供调用参数时，则会出错。真正的任务状态在回调存档服务时提供（信号与槽机制）。

存档类型

a) **批量调试存档** (SLSP_SETUP_ARC)
批量调试存档可实现：
- 在首次调试完成之后，使同类型机床的其他控制系统达到与首次调试后相同的状态。
- 在维修情况下（更换硬件后），可以花费极小的代价使一个新的控制系统达到起始状态。

b) **用户存档** (SLSP_NORMAL_ARC)
用户存档可由任意文件（例如零件程序、工件等）组成。其可用于对 NC 存储器和本地驱动器上的各个文件进行存档。

c) **PLC 升级存档** (SLSP_PLCUPGRADE_ARC)
PLC 升级存档只含有 PLC 的 SDB，用于 PLC 的升级。

d) **穿孔带存档** (SLSP_PAPERTAPE_ARC)
使用穿孔带存档可将文件以 ASCII 格式进行存档。此时应注意：
- 只能备份含可显示字符的文件，即使用文本编辑器创建的文件，而不能备份二进制数据。
- 穿孔带存档可使用文本编辑器进行创建或编辑。

注

穿孔带格式的说明请见 DOConCD 上“HMI-Embedded 操作手册”中的“服务操作区域”一章（关键字：穿孔带格式）。

访问授权

为了能使用存档服务的接口函数，应设置 S1（机床制造商）访问级别。 这可（例如）直接在代码中通过调用 PI 服务“_N_LOGIN_”或“_N_LOGOUT”来实现。

10.2 分步示例

概述

以下章节将分步介绍存档服务的不同应用区域。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例。

表 10-1：示例一览

应用	示例	章节
创建批量调试存档	slexarcreatereadarchive	10.2.2
创建用户存档	slexarcreatereadarchive	10.2.3
读入存档	slexarcreatereadarchive	10.2.4
监控存档服务的函数	slexarnotifystatus	10.2.5

此处只举例说明或描述与存档服务相关的内容。源文件的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

开发环境

建立开发环境所需的步骤在章节 5.3 中说明。

10.2.1 准备

概述

满足以下前提后，才能从自定义的项目或类中调用存档服务。

检查库

在项目设置中检查库 `slarchiveadapter.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `slarchiveadapter.lib`

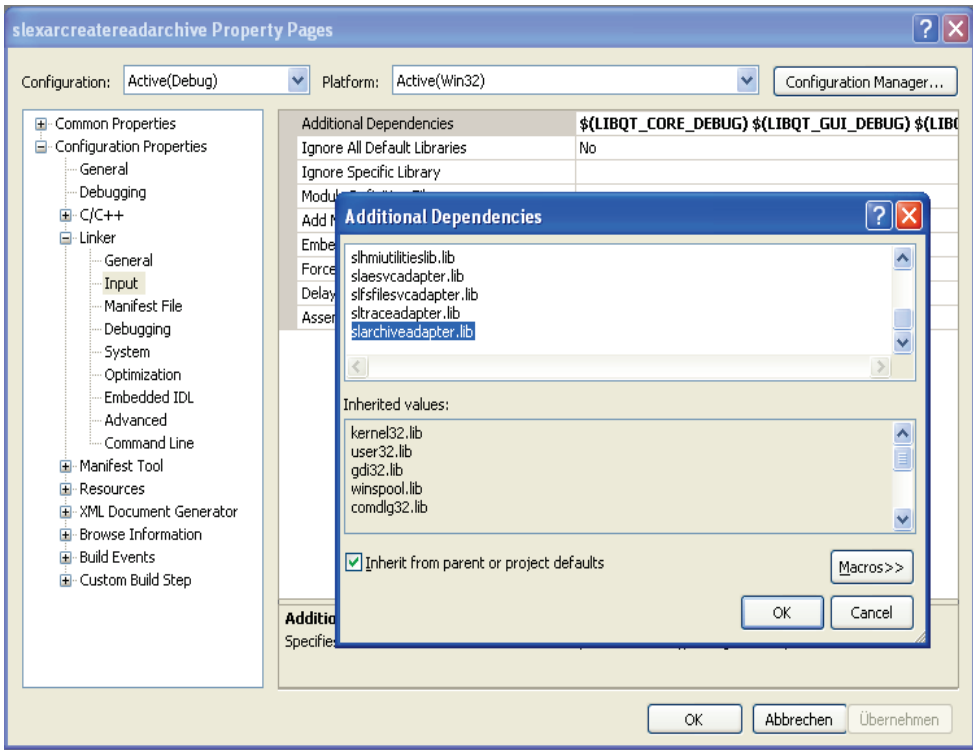


图 10-1:库“slarchiveadapter.lib”

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lslarchiveadapter
```

注

示例“SlExArNotifyStatus”（章节 10.2.5，“监控存档服务的函数”）无需完成下述从“头文件”起的准备工作，因为它使用的是类“SlArchiveSvcNotifier”。该示例要求的所有准备工作参见对应的“分步示例”。

头文件

要使用存档服务的类，需要加入头文件“slarchiveqsvc.h”，配置条目为：

```
#include "slarchiveqsvc.h"
```

创建 SlArchiveQSvc 对象

访问存档服务的接口需要使用 SlArchiveQSvc 类的对象。这些对象可作为私有的成员变量创建：

```
SlArchiveQSvc m_archiveSvr;
```

设置信号与槽的关联

存档服务通过信号 listCompleted 报告创建/读入存档的结果。信号 progress 可指明执行进度。这些信号由以下函数（槽）接收。

```
private slots:
   void listCompletedSlot(long lRequestID, SlSpErrorVec& rErrorVec,
                          long lRetCode);
   void progressSlot(long lRequestID, long lPercentage, QString& rstrInput);
```

注

通常情况下应使用这两个信号。其他所有可用信号参见 SlArchiveQSvc 接口的引用。

接着将成员函数（槽）与对应的存档服务的信号关联在一起。 比如：在示例 “SlExArCreateReadArchive”中，关联是在构造函数中完成的。

```
QObject::connect(&m archiveSvr,
                 SIGNAL(listCompleted(long, SlSpErrorVec&,long)),
                 this,
                 SLOT(listCompletedSlot(long, SlSpErrorVec&,long)));

QObject::connect(&m archiveSvr,
                 SIGNAL(progress(long, long, QString&)),
                 this,
                 SLOT(progressSlot(long, long, QString&)));
```

初始化 SlArchiveQSvc

在使用 SlArchiveQSvc 类的各个函数前，首先要初始化该类。为此要调用函数 init()。
如果 SlArchiveQSvc 对象是作为私有成员变量创建的，最好在
构造函数中进行初始化：

```
m_archiveSvr.init();
```

释放资源

不再需要 SlArchiveQSvc 对象时，需要调用函数 fini()。这样可以再次释放原先被占用的资源。如果 SlArchiveQSvc 对象是作为私有成员变量创建的，最好在析构函数中释放资源：

```
m_archiveSvr.fini();
```

10.2.2 创建批量调试存档

下面的示例分步介绍了创建批量调试存档的必要操作。 SINUMERIK Operate .net 软件包中相应的可执行示例为“SIExArCreateReadArchive”。 本章将介绍水平软键“create setup archive”的相关功能。

在示例中可在“archive type”下选择存档的类型。 通过“archive path”和“archive name”设置存档的存储位置。 异步调用的进度可在“archive protocol”中查看。状态栏显示调用成功。

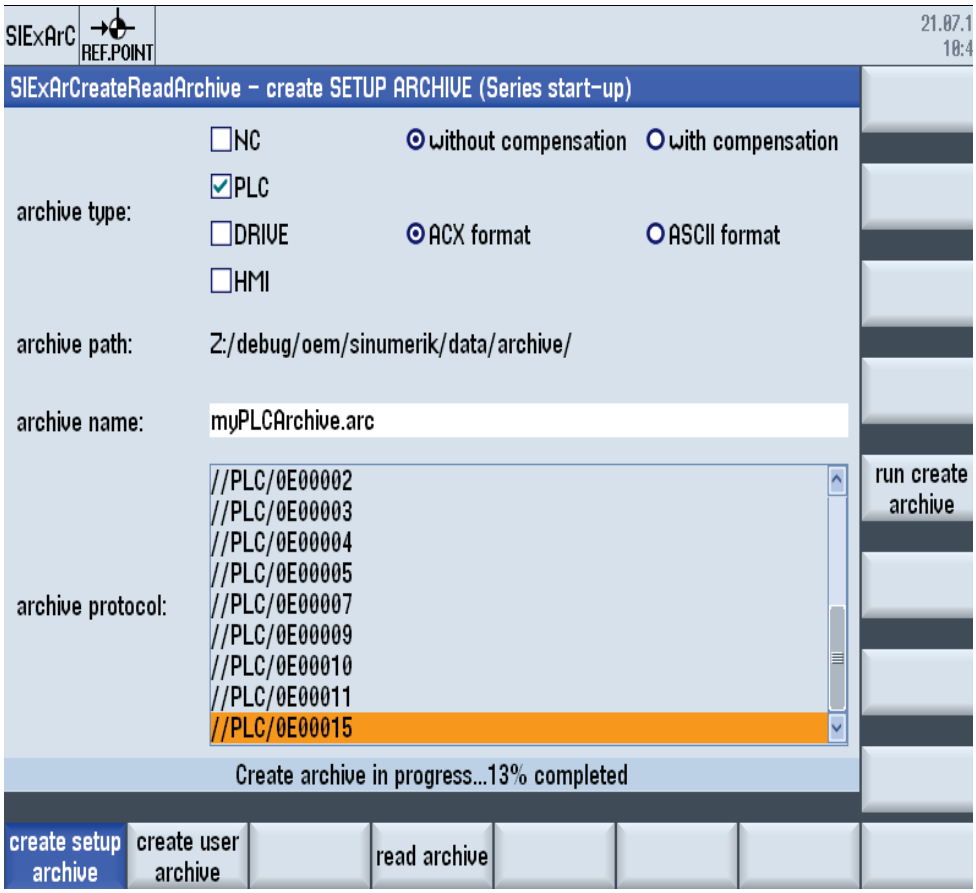


图 10-2: 示例 SIExArCreateReadArchive - 水平软键“create setup archive”

务必要足章节 10.2.1 列出的前提条件。

第 1 步

将需要存档的组件(NC, PLC, DRIVE, …)加入到 QStringList 中。

```
QStringList lstSelectedTypes;
...
lstSelectedTypes.append(SLARCHIVE_NCU);
lstSelectedTypes.append(SLARCHIVE_PLC);
lstSelectedTypes.append(SLARCHIVE_DRIVES);
...
```

注

在函数 `createArchiveInput_syn` 的引用中说明了所有组件关键字，比如：`SLARCHIVE_NCU`。

第 2 步

将创建的、由组件关键字组成的列表(`lstSelectedTypes`)传送给函数 `createArchiveInput_syn`。由此生成一张用于函数 `createArchiv` 的输入列表。

```
QStringList lstArchiveList;
long lRetVal = m archiveSvr.createArchiveInput syn(SLSP SETUP ARC,
                                                   lstSelectedTypes,
                                                   lstArchiveList);
```

第 3 步

向存档服务传送任务：从输入列表(`lstArchiveList`)中创建一份批量调试存档。返回的句柄 `m_lRequestId` 用于标识任务。

```
if ( 0 <= lRetVal )
{
    lRetVal = m_archiveSvr.createArchive(lstArchiveList,
                                         szFullArchiveName,
                                         SLSP SETUP ARC,
                                         QString::null,
                                         m_lRequestId);
}
```

第 4 步

调用立即返回。任务的执行进度在槽 `progressSlot` 中指明。在示例中，百分比进度在状态栏中输出，当前正在处理的任务/文件(`rstrInput`)在记录中输出。

需要检查它是否是正确的任务时，可以对比 `m_lRequestId` 与槽 `progressSlot` 传送的 `lRequestID`。

```
void SlExArCreateSetupArchiveForm::progressSlot(long lRequestID,
                                                long lPercentage,
                                                QString& rstrInput)
{
    if ( lRequestID == m_lRequestId )
    {
        // write progress to status bar
        setStatusBarElementText(CREATEARCHIVE PROGRESS.arg(lPercentage), 0);

        // write info to protocol
        ...
        m pProtocolList->addItem(rstrInput);
    }
}
```

第 5 步

调用了槽 listCompletedSlot 后，创建存档任务结束。

需要检查它是否是正确的任务时，可以对比 m_lRequestId
与槽 progressSlot 传送的 lRequestID。

```
void SlExArCreateSetupArchiveForm::listCompletedSlot(long lRequestID,
                                                     SlSpErrorVec&,
                                                     long lRetCode)
{
    if ( lRequestID == m_lRequestId )
    {
        if( 0 > lRetCode )
        {
            // Insert an error handling routine
        }
        else
        {
            // Archive successfully created
        }
    }
}
```

10.2.3 创建用户存档

下面的示例分步介绍了创建用户存档的必要操作。 SINUMERIK Operate .net 软件包中相应的可执行示例为“SIExArCreateReadArchive”。 本章将介绍水平软键“create user archive”的相关功能。

在示例中可在“data list”下选择要对哪些零件程序进行存档。 通过“archive path”和“archive name”设置存档的存储位置。 异步调用的进度可在“archive protocol”中查看。状态栏显示调用成功。

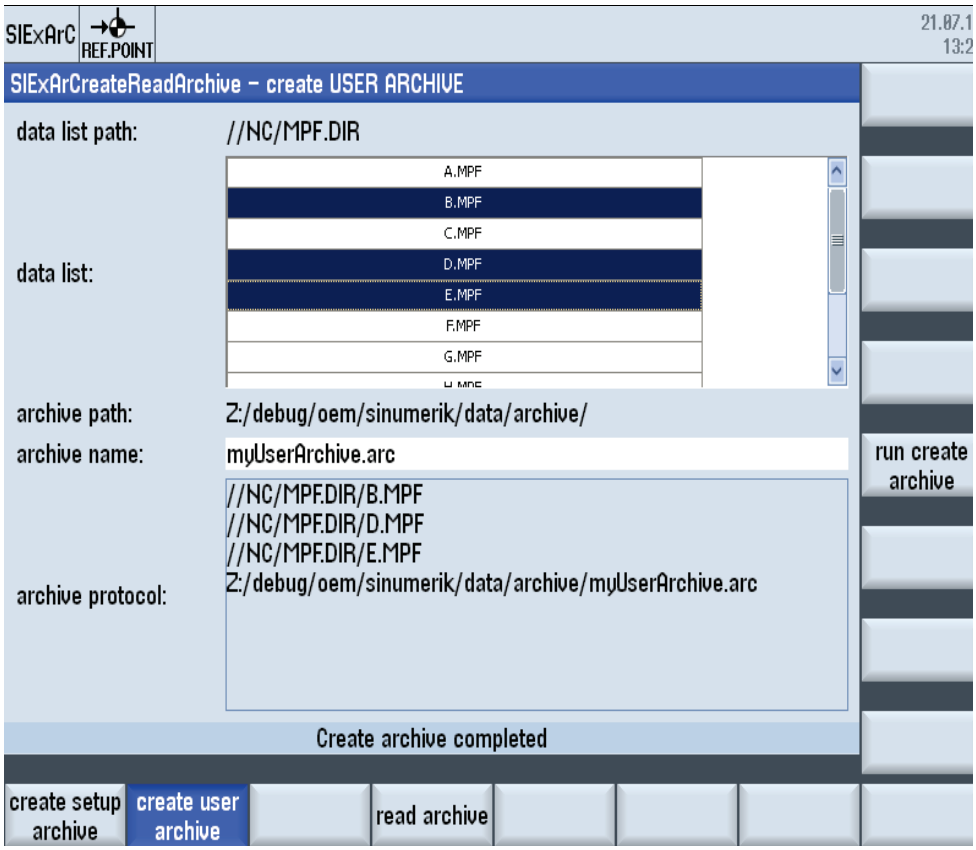


图 10-3: 示例 SIExArCreateReadArchive - 水平软键“create user archive”

务必要足章节 10.2.1 列出的前提条件。

第 1 步

将需要存档的文件（逻辑路径）加入到 QStringList 中。

```
QStringList lstArchiveInput;
...
lstArchiveInput.append("//NC/MPF.DIR/B.MPF");
lstArchiveInput.append("//NC/MPF.DIR/D.MPF");
lstArchiveInput.append("//NC/MPF.DIR/E.MPF");
...
```

第 2 步

向存档服务传送任务：从输入列表(`lstArchiveList`)中创建一份用户存档。返回的句柄 `m_lRequestId` 用于标识任务。

```
long lRetVal = m archiveSvr.createArchive(lstArchiveInput,
                                          szFullArchiveName,
                                          SLSP NORMAL ARC,
                                          QString::null,
                                          m_lRequestId);
```

第 3 步

调用立即返回。任务的执行进度在槽 `progressSlot` 中指明。在示例中，百分比进度在状态栏中输出，当前正在处理的任务/文件(`rstrInput`)在记录中输出。

需要检查它是否是正确的任务时，可以对比 `m_lRequestId` 与槽 `progressSlot` 传送的 `lRequestID`。

```
void SlExArCreateUserArchiveForm::progressSlot(long lRequestID,
                                               long lPercentage,
                                               QString& rstrInput)
{
    if ( lRequestID == m lRequestId )
    {
        // write progress to status bar
        setStatusBarElementText(CREATEARCHIVE_PROGRESS.arg(lPercentage), 0);

        // write info to protocol
        ...
        m_pProtocolList->addItem(rstrInput);
        ...
}
```

第 4 步

调用了槽 `listCompletedSlot` 后，创建存档任务结束。

需要检查它是否是正确的任务时，可以对比 `m_lRequestId` 与槽 `progressSlot` 传送的 `lRequestID`。

```
void SlExArCreateUserArchiveForm::listCompletedSlot(long lRequestID,
                                                    SlSpErrorVec&,
                                                    long lRetCode)
{
    if ( lRequestID == m_lRequestId )
    {
        if( 0 > lRetCode )
        {
            // Insert an error handling routine
        }
        else
        {
            // Archive successfully created
        }
    }
}
```

10.2.4 读入存档

下面的示例分步介绍了读入存档的必要操作。 SINUMERIK Operate .net 软件包中相应的可执行示例为“SlExArCreateReadArchive”。 本章将介绍水平软键“read archive”的相关功能。

在本例中，可以在“archive list”下选择需要读入的现有存档（批量调试存档、用户存档等）。 异步调用的进度可在“archive protocol”中查看。状态栏显示调用成功。

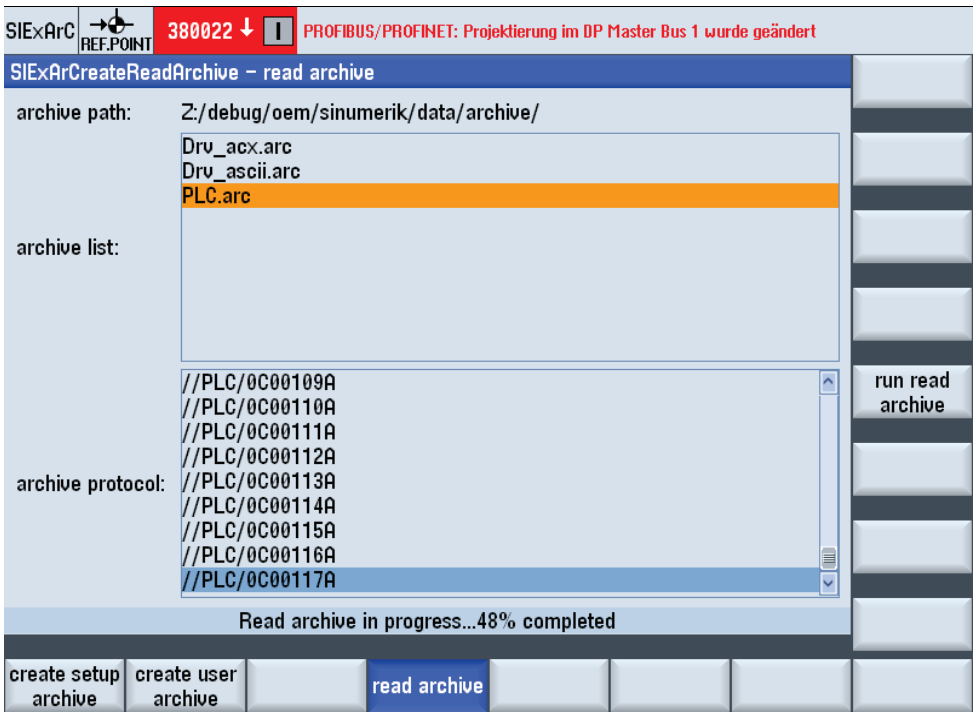


图 10-4: 示例 SlExArCreateReadArchive – 水平软键“read archive”

务必要足章节 10.2.1 列出的前提条件。

第 1 步

选择选项 SLARCHIVE_FORCE_ALL，在读入存档时不会输出任何确认提示。选择 SLARCHIVE_CHECK_ALL，执行所有检查。

```
ulong ulQuitCheckMode = SLARCHIVE_FORCE_ALL + SLARCHIVE_CHECK_ALL;
```

注

在函数 readArchive 的引用中说明了该参数的所有选项。

第 2 步

向存档服务传送任务：读入存档(szFullArchiveName)。返回的句柄 m_lRequestId 用于标识任务。

```
QString szFullArchiveName = "Z:/hmisl/oem/sinumerik/data/archive/PLC.arc"

long lRetVal = m_archiveSvr.readArchive(szFullArchiveName,
                                        ulQuitCheckMode,
                                        m_lRequestId);
```

第 3 步

调用立即返回。任务的执行进度在槽 progressSlot 中指明。在示例中，百分比进度在状态栏中输出，当前正在处理的任务/文件(rstrInput)在记录中输出。

需要检查它是否是正确的任务时，可以对比 m_lRequestId
与槽 progressSlot 传送的 lRequestID。

```
void SlExArReadArchiveForm::progressSlot(long lRequestID,
                                         long lPercentage,
                                         QString& rstrInput)
{
    if ( lRequestID == m_lRequestId )
    {
        // write progress to status bar
        setStatusBarElementText(READARCHIVE PROGRESS.arg(lPercentage), 0);

        // write info to protocol
        ...
        m_pProtocolList->addItem(rstrInput);
        ...
}
```

第 4 步

调用了槽 listCompletedSlot 后，存档读入任务结束。

需要检查它是否是正确的任务时，可以对比 m_lRequestId
与槽 progressSlot 传送的 lRequestID。

```
void SlExArReadArchiveForm::listCompletedSlot(long lRequestID,
                                              SlSpErrorVec&,
                                              long lRetCode)
{
    if ( lRequestID == m_lRequestId )
    {
        if( 0 > lRetCode )
        {
            // Insert an error handling routine
        }
        else
        {
            // Archive successfully created
        }
    }
}
```

10.2.5 监控存档服务的函数

下面的示例分步介绍了如何监控存档服务中正在执行的异步函数从而作出所需响应，比如对标准操作区域“调试”中的批量调试作出响应。SINUMERIK Operate .net 软件包中相应的可执行示例为“SlExArNotifyStatus”。

示例中显示了一张记录存档服务中所有异步函数调用（每个调用的开始和结束）的列表。

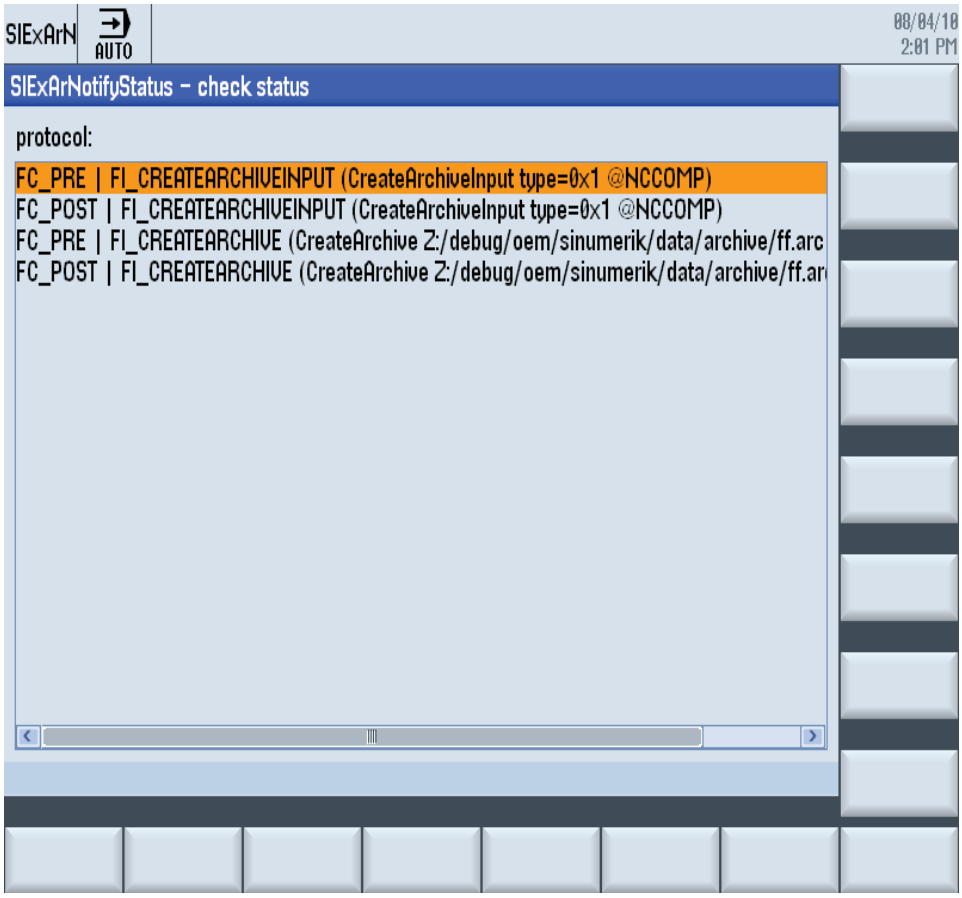


图 10-5:示例 SlExArNotifyStatus

要满足章节 10.2.1“准备”列出的部分前提条件（参见“提示”）。

第 1 步

加入 SlArchiveSvcNotifier 类的头文件：

```
#include "slarchivesvcnotifier.h"
```

第 2 步

定义从 SlArchiveSvcNotifierCallbackInterface 导出的回调类。

```
class SlExArNotifyStatusCallBack :publicSlArchiveSvcNotifierCallbackInterface
{
   ...
private:
   void notify(SlSpFunctionCall_Enum enumCallState,
               SlSpFunctionId_Enum enumFunctionId,
               const QString& rstrInfo);
   ...
};
```

第 3 步

创建以下私有成员变量。第一个变量是真正的接口类 SlArchiveSvcNotifier 的一个对象。最后一个变量是第 2 步中定义的 Callback 类的指针。

```
SlArchiveSvcNotifier        m_archiveSvcNotifier;
SlExArNotifyStatusCallBack* m_pCallback;
```

第 4 步

在激活存档服务的监控功能前，首先要初始化该类。初始化最好在构造函数中进行。

```
m_archiveSvcNotifier.init();
```

第 5 步

创建 Callback 类的一个对象。接着注册 Callback。

```
m pCallback = new SlExArNotifyStatusCallBack(this);
m_archiveSvcNotifier.registerCallback(m_pCallback, false);
```

第 6 步

现在将存档服务 Callback 类中的所有异步函数调用（每个调用的开头和末尾）通知给被覆盖的函数 notify。

```
void SlExArNotifyStatusCallBack::notify(SlSpFunctionCall Enum enumCallState,
                                        SlSpFunctionId_Enum enumFunctionId,
                                        const QString& rstrInfo)
{
   ...
   switch (enumCallState)
   {
      case SLSP_FC_PRE:szCallState = "FC_PRE"; break;
      case SLSP_FC_POST:szCallState = "FC_POST"; break;
   }
   ...
}
```

第 7 步

不再需要该功能时，需要注销 Callback。

```
m_archiveSvcNotifier.unregisterCallback();
```

第 8 步

最后释放被占用的资源。释放最好在析构函数中进行。

```
m_archiveSvcNotifier.fini();

// clear callback object
delete(m_pCallback);
m_pCallback = 0;
```

10.3 SlArchiveQSvc 引用

10.3.1 定义

概述

使用 SlArchiveQSvc 对象可以生成和读入不同的存档类型。

创建分为两步：
a) 首先创建一张列出所有需要加入到存档中的文件/目录和任务的列表。
b) 接着根据该列表创建存档。

在读入时存档的内容会解包到相应的路径下。

列表创建（步骤 a）这一任务既可以同步调用也可以异步调用。存档的生成和读取因耗时较长而只能进行异步调用。

同步调用此时会阻塞调用所在的线程，
直到任务完成。而在异步调用中，
SlArchiveQSvc 对象会在任务完成后利用“Qt 的信号与槽机制”发送信号。

10.3.2 初始化和终止函数

初始化(init)

初始化 SlArchiveQSvc 对象。该函数作为第一个函数调用。

表 10-2：init

long SlArchiveQSvc::init();	
参数	**含义**
返回值	函数的执行状态

示例：
见“SlExArCreateReadArchive”。

终止(fini)

释放 SlArchiveQSvc 对象占用的资源。该函数作为最后一个函数调用。

表 10-3：fini

long SlArchiveQSvc::fini();	
参数	**含义**
返回值	函数的执行状态

示例：
见“SlExArCreateReadArchive”。

10.3.3 生成存档

创建输入列表

通过函数 `createArchiveInput_syn` 和 `createArchiveInput` 可以分别同步创建和异步创建一个输入列表用于 `createArchive`。该输出列表指出了要对哪些内容进行存档。

表 10-4：createArchiveInput_syn

long SlArchiveQSvc::createArchiveInput_syn(SlSpArchiveType_Enum enumArchiveType, const QStringList& rstrSelectedFiles, QStringList& rstrlistArchiveInput, QString strNcuName = "");	
参数	含义
enumArchiveType	待创建的存档的类型： SLSP_NORMAL_ARC, SLSP_SETUP_ARC, SLSP_PLCUPGRADE_ARC
rstrSelectedFiles	需要创建存档的所有内容的列表。其中可以包括目录、文件绝对名（逻辑路径）和组件关键字。
rstrlistArchiveInput	指出需要存档的文件绝对名称或任务的列表。（返回值）
strNcuName	无含义（不传送任何内容）
返回值	创建输入列表的执行状态。

在异步调用中会在调用后返回一个句柄，用于标识调用。信号 `inputListCompleted` 在异步调用成功完成后发送，信号 `canceled` 在异步调用被取消后发送。

表 10-5：createArchiveInput

long SlArchiveQSvc::createArchiveInput(SlSpArchiveType_Enum enumArchiveType, const QStringList& rstrSelectedFiles, long& rlRequestID, QString strNcuName = "");	
参数	含义
enumArchiveType	待创建的存档的类型： SLSP_NORMAL_ARC, SLSP_SETUP_ARC, SLSP_PLCUPGRADE_ARC
rstrSelectedFiles	需要创建存档的所有内容的列表。其中可以包括目录、文件绝对名（逻辑路径）和组件关键字。
rlRequestID	用于标识异步调用的句柄(handle)。（返回值）
strNcuName	无含义（不传送任何内容）
返回值	函数的执行状态。

以下组件关键字可用于 rstrSelectedFiles：

表 10-6：组件关键字

关键字	描述
SLARCHIVE_NCU	所有 NC 数据
SLARCHIVE_NCUCOMP	所有 NC 数据+补偿数据 （也包含 SLARCHIVE_NCU）
SLARCHIVE_PLC	PLC 数据
SLARCHIVE_PLCUPGRADE	PLC 升级数据（仅 SDB）
SLARCHIVE_COMPILECYCLES	编译循环数据
SLARCHIVE_DRIVES	驱动数据（二进制格式）
SLARCHIVE_DRIVES_ASCII	驱动数据（ASCII 格式）
SLARCHIVE_HMI	HMI 数据

以下是一些只适用于特定存档类型(enumArchiveType)的组件关键字：

表 10-7：组件关键字和存档类型

存档类型	允许的关键字
SLSP_NORMAL_ARC	- SLARCHIVE_HMI - SLARCHIVE_COMPILECYCLES
SLSP_SETUP_ARC	- SLARCHIVE_NCU *或* SLARCHIVE_NCUCOMP - SLARCHIVE_PLC - SLARCHIVE_COMPILECYCLES - SLARCHIVE_DRIVES *或* SLARCHIVE_DRIVES_ASCII - SLARCHIVE_HMI
SLSP_PLCUPGRADE_ARC	- SLARCHIVE_PLCUPGRADE
SLSP_PAPERTAPE_ARC	（手动创建的输入列表）

! **重要提示**

请勿使用上表未列出的组件关键字，否则有可能无法正确创建存档或者创建的存档无法再次读入。

示例：
见“SlExArCreateReadArchive”。

创建存档

函数 createArchive 可用于创建存档。函数异步调用，立即返回。信号 listCompleted 在异步调用成功完成后发送，信号 canceled 在异步调用被取消后发送。信号 progress 可指明执行进度。

该函数也可以传送一条注释。该注释随后会显示在“调试”操作区“系统数据”的预览窗口中。如果希望在注释中加入创建人员的姓名，可使用关键字“creator:”，然后在“arg”中加入某语言的注释信息：

注释示例：

```
QString szComment = QString("creator:%1\n%2\n%3").arg("Max Mustermann")
                                                 .arg("some informations")
                                                 .arg("more informations");
```

该注释会如下显示在预览窗口中：

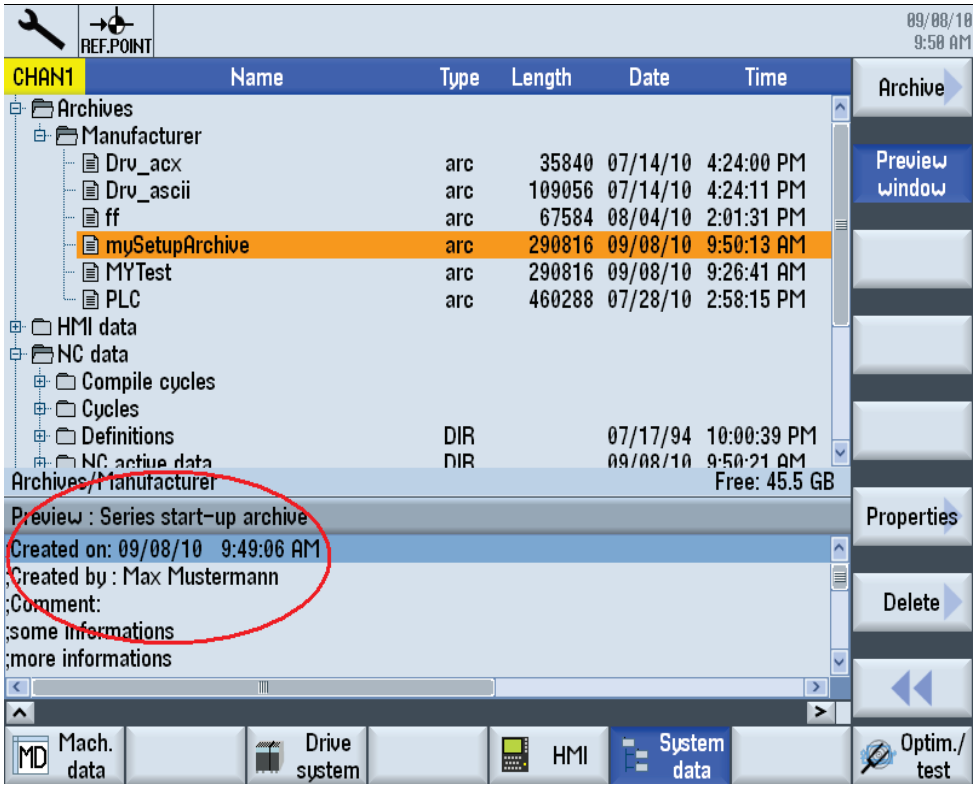


图 10-6:调试/系统数据的预览窗口

表 10-8：createArchive

long SlArchiveQSvc::createArchive(const QStringList& rstrlistArchiveInput, const QString& rstrFullArchiveName, SlSpArchiveType_Enum enumArchiveType, const QString& rstrComment, long& rlRequestID, QString strNcuName = "");	
参数	**含义**
rstrlistArchiveInput	包含了需要存档的文件的绝对名称或任务的列表（另见 `createArchiveInput_syn` 和 `createArchiveInput`）
rstrFullArchiveName	需要存档的文件的绝对名称。
enumArchiveType	待创建的存档的类型： SLSP_NORMAL_ARC, SLSP_SETUP_ARC, SLSP_PLCUPGRADE_ARC
rstrComment	显示在预览窗口中的注释。
rlRequestID	用于标识异步调用的句柄(handle)。（返回值）
strNcuName	无含义（不传送任何内容）
返回值	函数的执行状态。

示例：
见“SlExArCreateReadArchive”。

生成穿孔带存档

函数 `createPtArchive` 可用于生成穿孔带存档（仅 ASCII 格式的文件）。函数异步调用，立即返回。信号 `listCompleted` 在异步调用成功完成后发送，信号 `canceled` 在异步调用被取消后发送。信号 `progress` 可指明执行进度。

此时应注意：

- 只能备份含可显示字符的文件，即使用文本编辑器创建的文件，而不能备份二进制数据。
- 穿孔带存档可使用文本编辑器进行创建或编辑。

注

穿孔带格式的说明请见 DOConCD 上“HMI-Embedded 操作手册”中的“服务操作区域”一章（关键字：穿孔带格式）。

表 10-9：createPtArchive

```
long SlArchiveQSvc::createPtArchive(
                    const QStringList& rstrlistArchiveInput,
                    const QString& rstrFullArchiveName,
                    long& rlRequestID,
                    bool  bCRandNL = false,
                    SlSpPtMode_Enum enumMode = SLSP_PTMODE_DIN,
                    const QString& rstrFullIsoDirPath = QString::null,
                    QString strNcuName = "");
```

参数	含义
rstrlistArchiveInput	指出需要存档的文件绝对名称或任务的列表。
rstrFullArchiveName	需要存档的文件的绝对名称。
rlRequestID	用于标识异步调用的句柄(handle)。 （返回值）
bCRandNL	换行方式： true ➔ "\r\n" false ➔ "\n"
enumMode	穿孔带存档的格式 SLSP_PTMODE_DIN ➔ DIN 格式 SLSP_PTMODE_ISO ➔ ISO 格式
rstrFullIsoDirPath	无含义（不传送任何内容）
strNcuName	无含义（不传送任何内容）
返回值	函数的执行状态。

请勿使用函数 `createArchiveInput_syn` 和 `createArchiveInput` 。可以手动创建 `rstrlistArchiveInput`，步骤和“创建用户存档”分步示例中的“SlExArCreateReadArchive - 水平软键“create user archive”一致。

10.3.4 读入存档

读入存档

函数 readArchive 可用于读入存档。函数异步调用，立即返回。信号 listCompleted 在异步调用成功完成后发送，信号 canceled 在异步调用被取消后发送。信号 progress 可指明执行进度。

表 10-10：readArchive

long SlArchiveQSvc::readArchive(	const QString& rstrFullArchiveName, ulong ulQuitCheckMode, long& rlRequestID, QString strNcuName = "")
参数	**含义**
rstrFullArchiveName	需要读入存档的文件的绝对名称。
ulQuitCheckMode	检查和确认提示的方式。
rlRequestID	用于标识异步调用的句柄(handle)。 （返回值）
strNcuName	无含义（不传送任何内容）
返回值	函数的执行状态。

参数 ulQuitCheckMode 可用于确定以信号形式输出哪种确认提示(ask...) 。另外，它还可用于确定在读入存档前执行哪些检查。该参数相当于常数中的一个位字段，也就是说：将常数加在一起便可以选中多个选项。

表 10-11：ulQuitCheckMode 的方式

常量	描述
SLARCHIVE_FORCE_NOTHING	输出所有必要的确认提示。
SLARCHIVE_FORCE_EXECUTE	封锁确认提示 askReadArchive（信号）的输出。
SLARCHIVE_FORCE_OVERRIDE	封锁确认提示 askOverwrite（信号）的输出。
SLARCHIVE_FORCE_MEMORYRESET	封锁确认提示 askEvent（信号）的输出。
SLARCHIVE_FORCE_NOASKNEWABSNAME	封锁确认提示 askNewAbsFile（信号）的输出。
SLARCHIVE_FORCE_ALL	不输出确认提示。
SLARCHIVE_CHECK_ALL	执行所有检查。
SLARCHIVE_CHECK_NOVERSION	不检查 NC 版本和型号。 （没有选择该选项时，会检查 NC 版本和型号。如果 NC 版本或型号不匹配，会输出确认提示（信号）askNckVersionType。）
SLARCHIVE_CHECK_NOSUM	不检查校验和。
SLARCHIVE_CHECK_NOTHING	没有任何检查。

示例：
见“SlExArCreateReadArchive”。

应答函数

在读入存档时可能会输出确认提示，输出的具体提示取决于选项 `ulQuitCheckMode`。有两个应答函数可用于应答提示。

函数 `sendReply` 用于应答下列确认提示：

- `askReadArchive`
- `askOverwrite`
- `askNckVersionType`
- `askEvent`

表 10-12：sendReply

long SIArchiveQSvc::sendReply(long lRequestId, SISpReply_Enum enumReply);	
参数	含义
rlRequestID	用于标识异步调用的句柄(handle)。
enumReply	提供以下应答方法： SLSP_ANSWER_YES ➔是，仅针对该提示 SLSP_ANSWER_YESALL ➔是，以后自动采用相同的动作处理同类提示 SLSP_ANSWER_NO ➔否，仅针对该提示 SLSP_ANSWER_NOALL ➔否，以后自动采用相同的动作处理同类提示
返回值	函数的执行状态。

函数 `sendReplyNewAbsFile` 用于应答下列确认提示：

- `askNewAbsFile`

表 10-13：sendReplyNewAbsFile

long SIArchiveQSvc::sendReplyNewAbsFile(long lRequestId, SISpReply_Enum enumReply, const QString& rstrNewAbsFile)	
参数	含义
rlRequestID	用于标识异步调用的句柄(handle)。
enumReply	提供以下应答方法： SLSP_ANSWER_YES ➔是，仅针对该提示 SLSP_ANSWER_YESALL ➔是，以后自动采用相同的动作处理同类提示 SLSP_ANSWER_NO ➔否，仅针对该提示 SLSP_ANSWER_NOALL ➔否，以后自动采用相同的动作处理同类提示
rstrNewAbsFile	新的文件绝对名称。
返回值	函数的执行状态。

10.3.5 其他函数

取消任务

指定对应的句柄可以取消异步任务。成功取消后会发送信号 canceled。

表 10-14：cancel

long SlArchiveQSvc::cancel(long lRequestId);	
参数	含义
rlRequestID	用于标识异步调用的句柄(handle)。
返回值	函数的执行状态。

查询存档条目

函数 getArchiveEntries 可用于预览存档。该函数为穿孔带存档返回各个条目，为其他格式的存档返回目录。信号 entryListCompleted 在异步调用完成后发送，信号 canceled 在异步调用被取消后发送。

表 10-15：getArchiveEntries

long SlArchiveQSvc::getArchiveEntries(const QString& rstrFullArchiveName, long& rlRequestID);	
参数	含义
rstrFullArchiveName	存档的文件的绝对名称。
rlRequestID	用于标识异步调用的句柄(handle)。 （返回值）
返回值	函数的执行状态。

查询存档类型

函数 getArchiveType_syn 可用于查询存档类型。

表 10-16：getArchiveType

long SlArchiveQSvc::getArchiveType_syn(const QString& rstrFullArchiveName, SlSpArchiveType_Enum& renumArchiveType);	
参数	含义
rstrFullArchiveName	存档的文件的绝对名称。
renumArchiveType	返回的存档类型： SLSP_SETUP_ARC ➔批量调试存档 SLSP_NORMAL_ARC ➔用户存档 SLSP_PLCUPGRADE_ARC ➔ PLC 升级存档 SLSP_PAPERTAPE_ARC ➔穿孔带存档
返回值	查询任务的执行状态。

10.3.6 通知/信号

概述

上文 10.3 说明的函数会发送一些信号。这些信号在下文详细说明。

注

所有用作确认提示的信号(`askOverwrite`, `askReadArchive`, `askEvent`, `askNckVersionType`, `askNewAbsFile`)必须收到应答，否则不会继续处理存档。不需要输出各种确认提示时，可在读入存档(`readArchive`)时选择选项 `ulQuitCheckMode`。

---

listCompleted

该信号报告异步任务 `createArchive`, `createPtArchive` 或 `readArchive` 结束。

表 10-17：listCompleted

void listCompleted(	**long lRequestID,** **SlSpErrorVec& rErrorVec,** **long lRetCode);**
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rErrorVec	QVector 和对象，各个对象包含了一个错误返回值、原因的标识和可选的说明文本（取决于错误返回值）。
lRetCode	异步任务的执行状态。

表 10-18：SlSpErrorVec（选段）

```
typedef QVector<SlSpError> SlSpErrorVec;

class SlSpError
{
        long            m_lNr;
        SlSpSender_Enum         m_enumSender;
        QString         m_strText;
};
```

参数	含义
m_lNr	错误返回值
m_enumSender	错误原因的标识： SLSP_FROM_UNDEFINED ➔发送方不明 SLSP_FROM_SPSERVICE ➔存档服务 SLSP_FROM_FILESERVICE ➔文件服务 SLSP_FROM_DRIVESERVICE ➔驱动服务 SLSP_FROM_CAPSERVICE ➔Cap 服务
m_strText	可选错误文本。

示例：
见“SlExArCreateReadArchive”。

inputListCompleted

该信号报告异步任务 `createArchiveInput` 已结束。

表 10-19：inputListCompleted

void inputListCompleted(long lRequestID, QStringList& rstrInputList, long lRetCode);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rstrInputList	指出需要存档的文件绝对名称或任务的列表。
lRetCode	异步任务的执行状态。

entryListCompleted

该信号报告异步任务 `getArchiveEntries` 已结束。

表 10-20：entryListCompleted

void entryListCompleted(long lRequestID, SlSpArchiveType_Enum enumArchiveType, QStringList& rstrEntryList, long lRetCode);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
enumArchiveType	存档类型： SLSP_SETUP_ARC ➔批量调试存档 SLSP_NORMAL_ARC ➔用户存档 SLSP_PLCUPGRADE_ARC ➔ PLC 升级存档 SLSP_PAPERTAPE_ARC ➔穿孔带存档
rstrEntryList	预览条目的列表。
lRetCode	异步任务的执行状态。

canceled

该信号报告异步任务 `cancel` 结束或某个异步任务被取消。

表 10-21：canceled

void canceled(long lRequestID);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。

progress

该信号报告异步任务 `createArchive`, `createPtArchive` 和 `readArchive` 的执行进度。

表 10-22：progress

void progress(long lRequestID, long lPercentage, QString& rstrInput);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
lPercentage	任务执行进度，百分比值。
rstrInput	正在处理的文件/操作。

示例：
见“SlExArCreateReadArchive”。

info

该信号报告异步任务 `createArchiveInput`, `createArchive`, `createPtArchive` 和 `readArchive` 执行期间输出的附加信息。

表 10-23：info

void info(long lRequestID, SlSpInfo& rInfo);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rInfo	包含了 ID 号、发送方标识和可选说明文本（取决于 ID 号）的对象。

表 10-24：SlSpInfo（选段）

class SlSpInfo { long m_lNr; SlSpSender_Enum m_enumSender; QString m_strText; };	
参数	含义
m_lNr	ID 号： SLSP_INF_PROCESSING_DIR ➔指出正在处理的目录的信息。 SLSP_INF_NCK_VERSION_TYPE_FAILED ➔指出 NC 无法读出 NCK 版本和/或型号的信息。 SLSP_INF_BYTES_TRANSFERED ➔通过文件服务传送的文件的数量。
m_enumSender	发送方的标识： SLSP_FROM_UNDEFINED ➔发送方不明 SLSP_FROM_SPSERVICE ➔存档服务 SLSP_FROM_FILESERVICE ➔文件服务 SLSP_FROM_DRIVESERVICE ➔驱动服务 SLSP_FROM_CAPSERVICE ➔Cap 服务
m_strText	可选的信息文本。

error

该信号报告异步任务 `createArchiveInput`, `createArchive`, `createPtArchive` 和 `readArchive` 执行期间出现的错误。所有错误在任务结束时集中到 QVector 对象的信号 `listCompleted` 中。

表 10-25：error

void error(long lRequestID, SlSpError& rError);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rError	包含了一个错误返回值、原因的标识和可选的说明文本（取决于错误返回值）的对象。

SlSpError 类的定义参见信号 listCompleted。

askOverwrite

信号 askOverwrite 是确认提示，询问用户在读入存档时(readArchive)是否要覆盖已有的文件。该信号必须用 sendReply 应答。

表 10-26：askOverwrite

void askOverwrite(	long lRequestID, QString& rstrDestination);
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rstrDestination	需要覆盖的文件的绝对名称。

askReadArchive

信号 askReadArchive 是确认提示，询问用户是否读入存档(readArchive) 。该信号必须用 sendReply 应答。

注

信号 askReadArchive 仅在处理批量调试存档(SLSP_SETUP_ARC)或 PLC 升级存档(SLSP_PLCUPGRADE_ARC)时发送。

表 10-27：askReadArchive

void askReadArchive(	long lRequestID, SlSpArchiveType_Enum enumArchiveType, QString& rstrInfo);
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
enumArchiveType	存档类型： SLSP_SETUP_ARC ➔批量调试存档 SLSP_NORMAL_ARC ➔用户存档 SLSP_PLCUPGRADE_ARC ➔ PLC 升级存档 SLSP_PAPERTAPE_ARC ➔穿孔带存档
rstrInfo	存档名称

askEvent

信号 askEvent 用作通用的确认提示。该信号必须用 sendReply 应答。目前只定义了在读入存档时(readArchive)输出的确认提示。

表 10-28：askEvent

void askEvent(	long lRequestID, long lQuestionId);
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
lQuestionId	提示的 ID 号： SLSP_ASK_PROCESS_CFG_RESET_INI ➔询问是否要读入“CFG_RESET_INI”。

askNckVersionType

信号 askNckVersionType 是确认提示，询问用户是否在发现型号或版本不一致时仍读入存档(readArchive)。该信号必须用 sendReply 应答。

表 10-29：askNckVersionType

void askNckVersionType(long lRequestID, long lNr, QString& rstrInfo);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
lNr	不一致性错误的 ID 号： SLSP_INF_WRONG_NCK_VERSION ➔NCK 版本不一致。 SLSP_INF_WRONG_NCK_TYPE ➔NCK 型号不一致。 SLSP_INF_WRONG_NCK_VERSION_AND_TYPE ➔NCK 版本和 NCK 型号不一致。
rstrInfo	关于版本和型号的附加信息。

askNewAbsFile

信号 `askNewAbsFile` 是确认提示，询问用户在读入存档时(`readArchive`)是否要修改文件名称或文件路径。该信号必须用 `sendReplyNewAbsFile` 应答。

表 10-30：askNewAbsFile

void askNewAbsFile(long lRequestID, QString& rstrCurAbsFile);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rstrCurAbsFile	需要修改的文件的绝对名称。

modifiedGUDs

信号 `modifiedGUDs` 返回一张在读入存档后(`readArchive`)经过修改的 GUD 文件的列表。

表 10-31：modifiedGUDs

void modifiedGUDs(long lRequestID, QStringList& rstrGudList);	
参数	含义
lRequestID	用于标识异步调用的句柄(handle)。
rstrGudList	经过修改的 GUD 定义文件的列表。

10.4 SlArchiveSvcNotifier 引用

10.4.1 定义

概述

使用 SlArchiveSvcNotifier 对象可监控存档服务中正在执行的异步函数从而作出所需响应，比如对标准操作区域“调试”中进行的批量调试作出响应。

10.4.2 初始化和终止函数

初始化(init)

初始化 SlArchiveSvcNotifier 对象。该函数作为第一个函数调用。

表 10-32：init

long SlArchiveSvcNotifier::init();	
参数	含义
返回值	函数的执行状态

示例：
在示例“SlExArNotifyStatus”中使用。

终止(fini)

释放 SlArchiveSvcNotifier 对象占用的资源。该函数作为最后一个函数调用。

表 10-33：fini

long SlArchiveSvcNotifier::fini();	
参数	含义
返回值	函数的执行状态

示例：
在示例“SlExArNotifyStatus”中使用。

10.4.3 注册/注销 Callback

注册 Callback

注册 SlArchiveSvcNotifier 对象的 Callback。

表 10-34：registerCallback

long SlArchiveSvcNotifier::registerCallback(SlArchiveSvcNotifierCallbackInterface* pCallBack, bool bAsync);	
参数	含义
pCallBack	数据类型为 SlArchiveSvcNotifierCallbackInterface 的对象的指针。
bAsync	无含义（始终传送 FALSE）
返回值	函数的执行状态

示例：

在示例“SlExArNotifyStatus”中使用。

注销 Callback

删除 Callback 的关联。

表 10-35：unregisterCallback

long SlArchiveSvcNotifier::unregisterCallback(void);	
参数	含义
返回值	函数的执行状态

示例：
在示例“SlExArNotifyStatus”中使用。

10.4.4 通知

notify

报告 Callback 对象中正在处理的存档服务函数（数据类型 `SlArchiveSvcNotifierCallbackInterface`）的状态变化。

注

Callback(`notify`)既可以在执行自定义的存档服务函数时触发，也可以在标准操作区“调试”中进行批量调试时触发。

表 10-36：notify

void notify(	SlSpFunctionCall_Enum enumCallState, SlSpFunctionId_Enum enumFunctionId, const QString& rstrInfo);
参数	含义
enumCallState	状态： SLSP_FC_PRE ➔存档服务的一个函数已经启动。 SLSP_FC_POST ➔存档服务的一个函数已经结束。
enumFunctionId	指定启动或结束了存档服务的哪个函数： SLSP_FI_NONE SLSP_FI_CREATEARCHIVEINPUT SLSP_FI_CREATEARCHIVE SLSP_FI_CREATEARCHIVEFROMSELECTION SLSP_FI_CREATEPLCUPGARCHIVE SLSP_FI_READARCHIVE SLSP_FI_CONVERTTREETOARCHIVE SLSP_FI_CONVERTARCHIVETOTREE SLSP_FI_READARCHIVETREE SLSP_FI_CREATEARCHIVETREE SLSP_FI_CREATEARD SLSP_FI_MERGEARD SLSP_FI_GETARCHIVEENTRIES SLSP_FI_DELETETREEENTRIES SLSP_FI_ANALYSEJOBLIST SLSP_FI_EXECUTEJOBLIST SLSP_FI_LOADPTETOPLC SLSP_FI_CREATEPTARCHIVE
rstrInfo	附加信息。

示例：
在示例“SlExArNotifyStatus”中使用。

10.5 配置文件“slsp.ini”

概述

配置文件“slsp.ini”可用于修改存档服务的属性。原则上您无需创建这样一份配置文件，因为所有出厂值已经足够满足大部分应用的需要。

但是如果您需要修改出厂值，便需要创建一份名为“slsp.ini”的文本文件，在其中设置所需参数。接着将文件复制到以下一个目录中：

- **<安装路径>/user/sinumerik/hmi/cfg**
- **<安装路径>/oem/sinumerik/hmi/cfg**

值在重启 SINUMERIK Operate 后生效。

“Time”段中的参数

表 10-37： “slsp.ini”文件“Time”段中的参数

参数	描述
ProgressSleepTime	发送信号 progress 后的休眠时间。 ➔缺省值：0 毫秒
InfoSleepTime	发送信号 info 后的休眠时间。 ➔缺省值：0 毫秒
ErrorSleepTime	发送信号 error 后的休眠时间。 ➔缺省值：0 毫秒

示例：

```
[Time]
ProgressSleepTime=10
InfoSleepTime=10
ErrorSleepTime=10
```

有些情况下信号 progress 不会返回参数 lPercentage 100，因此最好将上述三个参数都设为 10 毫秒，如上例所示。

“SetupArc”段中的参数

表 10-38： “slsp.ini”文件“SetupArc”段中的参数

参数	描述
PlcStopWhileReadingSetup	***True***：PLC 在读入批量调试存档 (SLSP_SETUP_ARC)时暂停。 ***False***：PLC 只有在读入 PLC 条目时暂停。 ➔缺省值：False （PLC200 和 PLC319 上没有含义）

示例：

```
[SetupArc]
PlcStopWhileReadingSetup=true
```

11

11 驱动机床数据服务

本章主要内容

本章主要介绍 SINUMERIK Operate 编程包中隶属于服务组件的驱动机床数据服务。驱动机床数据服务可用于访问机床数据、设定数据、GUD 数据和驱动数据。

另外，它还可用于查询参数名称，以便在 CAP 服务中使用参数（见第 5 章）。

11.1 引言

11.1.1 类模型

概述

驱动机床数据服务的类模型主要由以下的类组成：

- SlQMd
- SlQMdDrvObject

SlQMd 类

通过 SlQMd 对象可以读/写机床数据、设定数据、GUD 数据和驱动数据。此处的驱动数据指的是 Sinamics 参数，这些参数不由 NCK 的 BTSS 接口提供。

所有访问函数同步调用，即阻塞调用所在的线程，直到任务完成。

只存在默认构造函数。SlQMd 对象无法
被复制或赋值。

注

SlQMd 对象既可以在 Qt 主线程中使用也可以在任何一个辅助线程中使用。

SlQMdDrvObject 类

用于保存驱动数据配置的数据结构。

11.1.2 术语解释

同步调用

同步调用在任务执行完后才会返回，即当前线程会被一直阻塞。这会干扰事件处理，例如在同步调用时阻止输入和显示。

单个变量调用

在单个调用时，只对一个变量进行读/写访问。有以下可用的单个调用：

表 11-1：单个调用

SlQMd 调用	描述
readNcData	同步读取一个 NC 变量（机床数据、设定数据和 GUD 数据）
writeNcData	同步写入一个 NC 变量（机床数据、设定数据和 GUD 数据）
readDrvData	同步读取一个驱动变量
writeDrvData	同步写入一个驱动变量

多变量调用

在多变量调用时，可访问多个变量。如果访问的多个 NC 变量，这些变量必须位于相同区域，具有相同的 BTSS 接口和相同的 GUD 编号。如果访问的是驱动数据，这些数据必须具有相同的驱动句柄。有以下可用的多变量调用：

表 11-2：多变量调用

SIQCap 调用	描述
readNcData	同步读取多个 NC 变量
writeNcData	同步写入多个 NC 变量
readDrvData	异步读取多个驱动变量
writeDrvData	异步写入多个驱动变量

注

在单变量调用和多变量调用中，真正的函数名称是一样的。区别只是在于函数的叠加。

数组访问

对于一些变量，可在其变量路径下定义多个连续的变量。这称之为数组访问。

数组访问加快了数据访问并可因此提高整个系统的速度，因为通讯时间被大大缩短了。

比如要访问三个连续的机床数据：

```
"$MN_AXCONF_MACHAX_NAME_TAB[0,2]"
```

11.2 分步示例

概述

以下章节将分步介绍驱动机床数据服务的不同应用区域。每个“分步示例”在 SINUMERIK Operate 编程包都有可执行的示例。

表 11-3：示例一览

应用	示例	章节
读/写一个 NC 变量	slexmdreadwritencdata（水平软键 1）	11.2.2
读/写多个 NC 变量	slexmdreadwritencdata（水平软键 2）	11.2.3
读/写驱动数据	slexmdreadwritedrvdata（水平软键 1）	11.2.4
读/写驱动数据（备选方法）	slexmdreadwritedrvdata（水平软键 2）	11.2.5

此处只举例说明或描述与驱动机床数据服务相关的内容。源文件的注释中还含有其他更加详细的信息（例如图片结构、输出等）。

另外，在 NC 变量的示例中还展示了如何查询用于 CAP 服务的变量名称。

开发环境

建立开发环境所需的步骤在章节 5.3 中说明。

11.2.1 准备

概述

满足以下前提后，才能从自定义的项目或类中调用驱动机床数据服务。

检查库

在项目设置中检查库 `slmd.lib` 是否已添加到 linker 中。执行以下操作：

1. 菜单“Project”
2. 菜单项“[Project name] properties...”
3. 浏览到“Configurations Properties / Linker / Input”。
4. 查看“Additional Dependencies”下的条目 `slmd.lib`

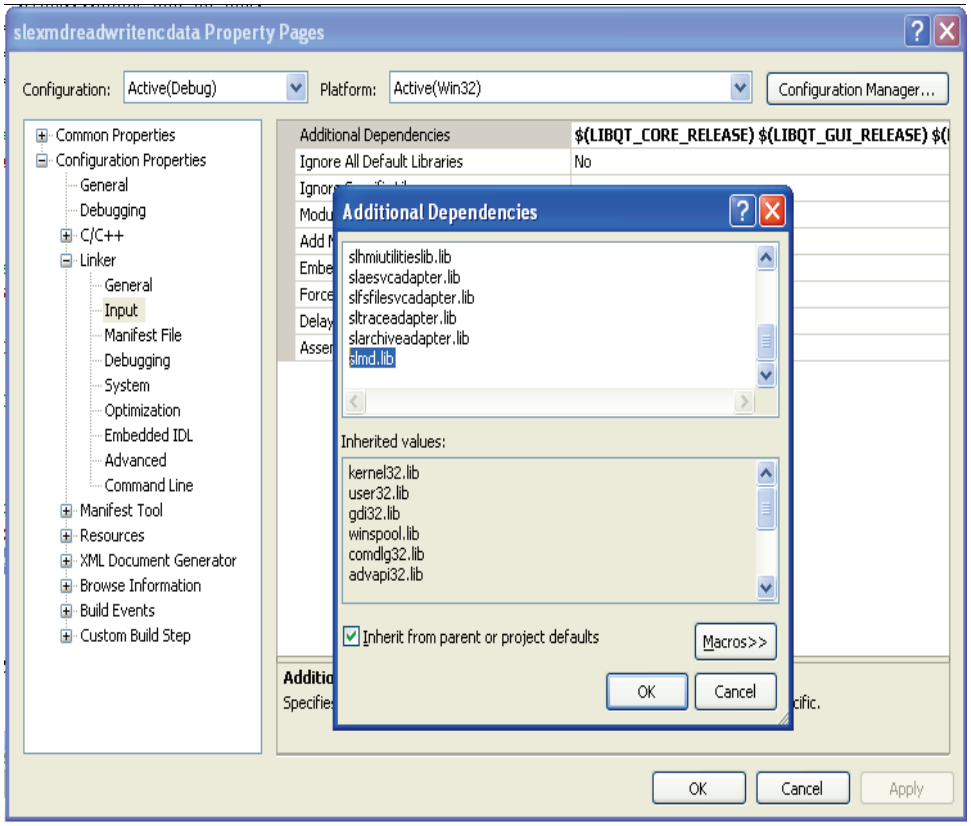


图 11-1:库“slmd.lib”

为确保与嵌入式 Linux 系统和 Pro 文件的兼容性，库名称为：

```
-lslmd
```

头文件

要使用驱动机床数据服务的类需要加入头文件“slqmd.h”，配置条目为：

```
#include "slqmd.h"
```

创建 SlQMd 对象

访问驱动机床数据服务的接口需要使用 SlQMd 类的对象。这些对象可作为私有的成员变量创建：

```
SlQMd m_objSlQMd;
```

11.2.2 读/写一个 NC 变量

下面的示例分步展示了如何读/写一个机床数据、设定数据或 GUD 数据。SINUMERIK Operate .net 软件包中相应的可执行示例为“SlExMdReadWriteNcData”。 本章将介绍水平软键“single access”的相关功能。

在示例中可以读/写当前选中的变量，可以通过“get actual linkItem”查询用于 CAP 服务的变量的名称。所有数据都可通过诸如双击的方式修改，以便测试其他变量。状态栏会显示

任务的执行状态。

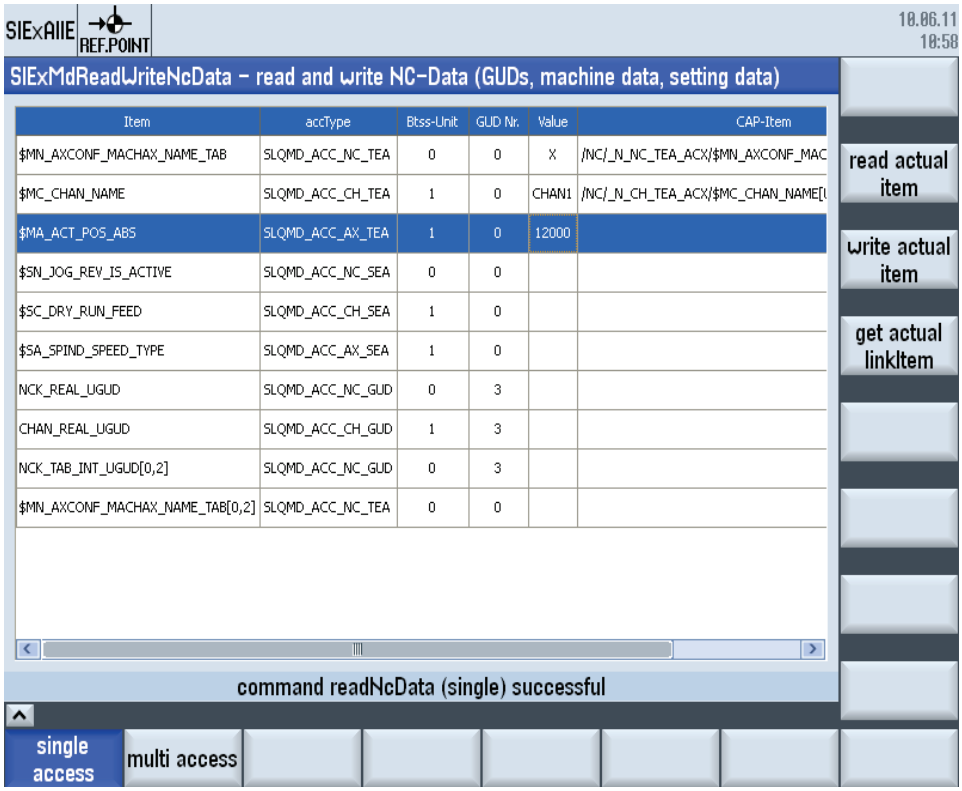


图 11-2: 示例 SIExMdReadWriteNcData – 水平软键“single access”

务必要满足章节 11.2.1“准备”列出的前提条件。

第 1 步

列出读/写任务需要使用的参数，查询用于 CAP 服务的变量的名称。

```
QString szItem                = "$MN_AXCONF_MACHAX_NAME_TAB[0]";
SlQMd::SlQMdAccEnum eAccType = SlQMd::SLQMD_ACC_NC_TEA;
long lBtssUnit                = 0;
long lGudNr                   = 0;
```

注

在引用中说明了所有有效的类型(`SlQMdAccEnum`)。

第 2 步

读通用机床数据“`$MN_AXCONF_MACHAX_NAME_TAB[0]`”，将读出的值写入 QString `szValue` 中。该任务的执行状态写入变量 `lRetVal` 中。

```
QString szValue     = QString::null;
long lRetVal = m_objSlQMd.readNcData(eAccType, lBtssUnit,
                                     szItem, szValue, lGudNr);
```

第 3 步

查看读任务的执行状态。

```
if ( 0 == lRetVal )
{
    // szValue 包含了读出的值，用于后续处理
}
else
{
    // 插入故障处理
}
```

第 4 步

将 szValue 的内容（“X”）写入到通用机床数据“$MN_AXCONF_MACHAX_NAME_TAB[0]”中。该任务的执行状态写入变量 lRetVal 中。

```
QString szValue = "X";
long lRetVal = m_objSlQMd.writeNcData(eAccType, lBtssUnit,
                                      szItem, szValue, lGudNr);
```

第 5 步

查看写任务的执行状态。

```
if( 0 != lRetVal )
{
    // 插入故障处理
}
```

第 6 步

查询用于 CAP 服务的变量的名称。

```
QStringList szLinkItems;
long lRetVal = m_objSlQMd.getNcDataLinkItem(eAccType, lBtssUnit,
                                  QStringList(szItem), szLinkItems, lGudNr);
```

第 7 步

查看任务的执行状态。

```
if ( 0 == lRetVal )
{
    QString szCapItem = szLinkItems[0];
    ...
}
else
{
    // 插入故障处理
}
```

11.2.3 读/写多个 NC 变量

下面的示例分步展示了如何读/写多个机床数据、设定数据或 GUD 数据。在执行这种多变量调用时，变量必须位于相同的区域内，具有相同的 BTSS 单位和 GUD 号。SINUMERIK Operate .net 软件包中相应的可执行示例为“SIExMdReadWriteNcData”。 本章将介绍水平软键“multi access”的相关功能。

在示例中可以读/写列表中显示的所有变量，可以通过“get all linkItem”查询用于 CAP 服务的所有变量的名称。所有数据都可通过诸如双击的方式修改，以便测试其他变量。状态栏显示调用成功。

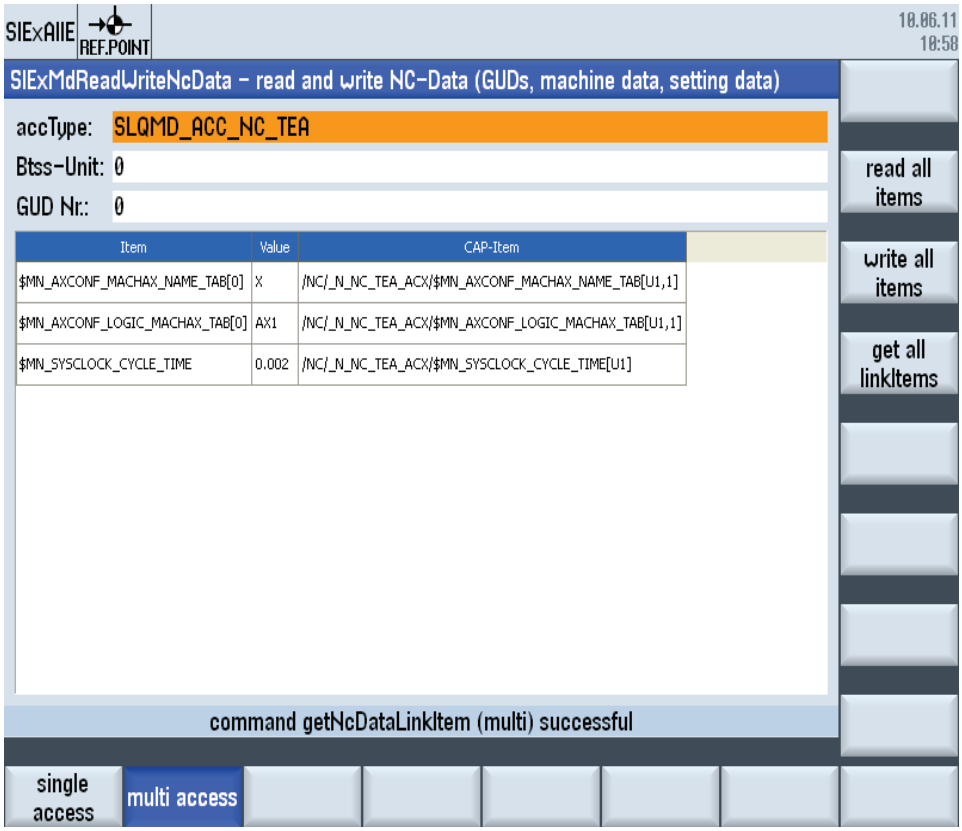


图 11-3: 示例 SIExMdReadWriteNcData – 水平软键“multi access”

务必要满足章节 11.2.1“准备”列出的前提条件。

第 1 步

列出读/写任务都需要使用的参数，查询用于 CAP 服务的变量的名称。所有变量的该参数必须相同。

```
SlQMd::SlQMdAccEnum eAccType = SlQMd::SLQMD ACC NC TEA;
long lBtssUnit               = 0;
long lGudNr                  = 0;
```

注

在引用中说明了所有有效的类型(`SlQMdAccEnum`)。

第 2 步

在 QStringList 中列出变量。

```
QStringList lstItems;
lstItems.append("$MN_AXCONF_MACHAX_NAME_TAB[0]");
lstItems.append("$MN_AXCONF_LOGIC_MACHAX_TAB[0]");
lstItems.append("$MN_SYSCLOCK_CYCLE_TIME");
```

第 3 步

读出 QStringList`lstItems` 中指定的所有变量，将读出的值写入到 QVector<QVariant> `vntValue` 中。所有变量读任务的执行状态写入变量 `lRetVal` 中。

```
QVector<QVariant> vntValue;
long lRetVal = m_objSlQMd.readNcData(eAccType, lBtssUnit,
                                     lstItems, vntValue, lGudNr);
```

第 4 步

查看读任务的执行状态。可以按照索引号来查看读出的单个值。

```
if ( 0 == lRetVal )
{
    QString szValue1 = vntValue[0].toString();
    QString szValue2 = vntValue[1].toString();
    QString szValue3 = vntValue[2].toString();
}
else
{
    // 插入故障处理
}
```

在出错时（返回值<> 0）可以检查单个变量的任务是否成功执行，方式如下：

```
bool bOkay = vntValue[0].isValid();
```

第 5 步

在 QStringList 中列出需要写入的各个变量值。

```
QStringList lstValues;
lstValues.append("X");
lstValues.append("AX1");
lstValues.append("0.002");
```

第 6 步

向 QStringList`lstItems` 中列出的变量写入 QStringList `lstValues` 的值。所有变量读任务的执行状态写入变量 `lRetVal` 中。

```
long lRetVal = m_objSlQMd.writeNcData(eAccType, lBtssUnit,
                                      lstItems, lstValues, lGudNr);
```

第 7 步

查看写任务的执行状态。

```
if( 0 != lRetVal )
{
```

```
    // 插入故障处理
}
```

第 8 步

为 QStringList`lstItems` 中列出的所有变量查询用于 CAP 服务的变量的名称。

```
QStringList szLinkItems;
long lRetVal = m_objSlQMd.getNcDataLinkItem(eAccType, lBtssUnit, lstItems,
                                            szLinkItems, lGudNr);
```

第 9 步

查看任务的执行状态。可以按照索引号来查看单个变量名称。

```
if ( 0 == lRetVal )
{
    QString szCapItem1 = szLinkItems[0];
    QString szCapItem2 = szLinkItems[1];
    QString szCapItem3 = szLinkItems[2];
    ...
}
else
{
    // 插入故障处理
}
```

11.2.4 读/写驱动数据

下面的示例分步展示了如何读/写一个驱动数据。 SINUMERIK Operate .net 软件包中相应的可执行示例为“SlExMdReadWriteDrvData”。 本章将介绍水平软键“type 1”的相关功能。

在示例中读取的是驱动配置。您可以根据选中的驱动对象选择一根有效的轴，然后读/写所需参数（不以字母开头）。状态栏会显示任务的执行状态。

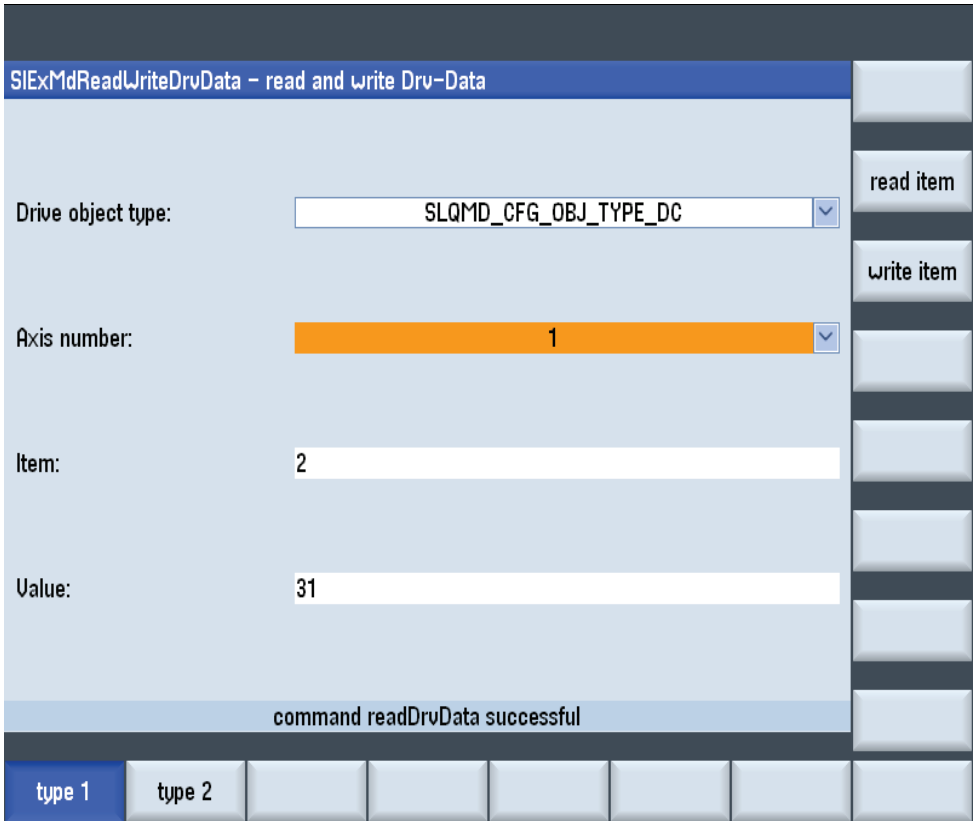


图 11-4: 示例 SIExMdReadWriteDrvData – 水平软键“type 1”

务必要满足章节 11.2.1“准备”列出的前提条件。

第 1 步

首先读取驱动配置。该任务的执行状态写入变量 `lRetVal` 中。

```
QVector<SlQMdDrvObject> drvObjects;
SlQMd::SlQMdDrvObjTypeEnum eDrvObjType = SlQMd::SLQMD_CFG_OBJ_TYPE_DC;
...
long lRetVal = m_objSlQMd.getDrvObjectList(eDrvObjType, drvObjects);
```

第 2 步

查看任务的执行状态。

```
if ( 0 == lRetVal )
{
    // 现在所有有效的轴可以查看
       返回的对象
    for ( int nIndex = 0; nIndex < drvObjects.count(); nIndex++ )
    {
        long lAxisNumber = drvObjects[nIndex].m_lAxNr;
        ...
    }
}
else
{
    // 插入故障处理
}
```

第 3 步

从所需轴号中得出对应驱动的唯一的一个句柄。

```
long lDrvHandle = m_objSlQMd.getDrvHandle(lAxisNumber);
```

第 4 步

读驱动数据“p2”，将读出的值写入 QVariant `vntValue` 中。该任务的执行状态写入变量 `lRetVal` 中。

```
QVariant vntValue;
QString szItem   = QString("2"); //p2
long lRetVal     = m_objSlQMd.readDrvData(lDrvHandle, szItem, vntValue);
```

第 5 步

查看读任务的执行状态。

```
if ( 0 == lRetVal )
{
    // vntValue 包含了读出的值，用于后续处理
}
else
{
    // 插入故障处理
}
```

第 6 步

将 szValue 的内容（“43”）写入到驱动数据“p2”中。该任务的执行状态写入变量 `lRetVal` 中。

```
QString szValue = "43";
long lRetVal= m_objSlQMd.writeDrvData(lDrvHandle, szItem, szValue);
```

第 7 步

查看写任务的执行状态。

```
if( 0 != lRetVal )
{
    // 插入故障处理
}
```

11.2.5 读/写驱动数据（备选方法）

下面的示例分步展示了如何读/写一个驱动数据。 SINUMERIK Operate .net 软件包中相应的可执行示例为“SlExMdReadWriteDrvData”。 本章将介绍水平软键“type 2”的相关功能。

在示例中读取的是驱动配置。 您可以根据选中的驱动对象选择一个有效的数据组，数据组由总线地址、从站地址和驱动对象 ID(DoId)组成，然后读/写所需参数（不以字母 p 开头）。状态栏会显示任务的执行状态。

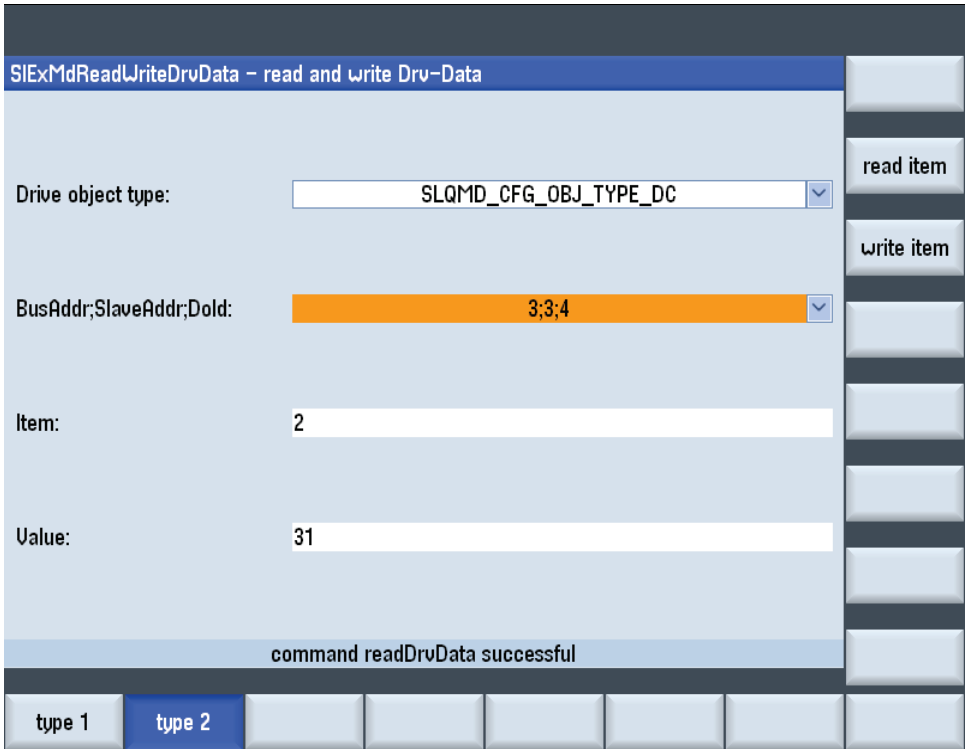


图 11-5: 示例 SIExMdReadWriteDrvData – 水平软键“type 2”

务必要满足章节 11.2.1“准备”列出的前提条件。

第 1 步

首先读取驱动配置。该任务的执行状态写入变量 `lRetVal` 中。

```
QVector<SlQMdDrvObject> drvObjects;
SlQMd::SlQMdDrvObjTypeEnum eDrvObjType = SlQMd::SLQMD_CFG_OBJ_TYPE_DC;
...
long lRetVal = m_objSlQMd.getDrvObjectList(eDrvObjType, drvObjects);
```

第 2 步

查看任务的执行状态。

```
if ( 0 == lRetVal )
{
    // 现在，所有由总线地址、从站地址和
       驱动对象 ID(DoId)组成的有效数据组可以参见
       返回的对象
    for ( int nIndex = 0; nIndex < drvObjects.count(); nIndex++ )
    {
        long lBusAddr   = drvObjects[nIndex].m_lBusAddr;
        long lSlaveAddr = drvObjects[nIndex].m_lSlaveAddr;
        long lDoId      = drvObjects[nIndex].m_lDoId;
        ...
    }
}
else
{
    // 插入故障处理
}
```

第 3 步

从所需数组中得出对应驱动的唯一的一个句柄。

```
long lDrvHandle = m_objSlQMd.getDrvHandle(lBusAddr, lSlaveAddr, lDoId);
```

第 4 步

读驱动数据“p2”，将读出的值写入 QVariant `vntValue` 中。该任务的执行状态写入变量 `lRetVal` 中。

```
QVariant vntValue;
QString szItem   = QString("2"); //p2
long lRetVal     = m_objSlQMd.readDrvData(lDrvHandle, szItem, vntValue);
```

第 5 步

查看读任务的执行状态。

```
if ( 0 == lRetVal )
{
    // vntValue 包含了读出的值，用于后续处理
}
else
{
    // 插入故障处理
}
```

第 6 步

将 szValue 的内容（“43”）写入到驱动数据“p2”中。该任务的执行状态写入变量 `lRetVal` 中。

```
QString szValue = "43";
long lRetVal= m_objSlQMd.writeDrvData(lDrvHandle, szItem, szValue);
```

第 7 步

查看写任务的执行状态。

```
if( 0 != lRetVal )
{
    // 插入故障处理
}
```

11.3 SIQMd 引用

11.3.1 定义

概述

通过 SIQMd 对象可以读/写机床数据、设定数据、GUD 数据和驱动数据。此处的驱动数据指的是 Sinamics 参数，这些参数不由 NCK 的 BTSS 接口提供。

所有访问函数同步调用，即阻塞调用所在的线程，直到任务完成。

只存在默认构造函数。SIQMd 对象无法
被复制或赋值。

AccTypen

访问机床数据、设定据和 GUD 时数据时需要使用和 NC 变量对应的 AccType。下表列出了可使用的 AccTypen。

表 11-4：有效的 AccTypen - 机床数据

机床数据	描述
SLQMD_ACC_BT_TEA	显示机床数据
SLQMD_ACC_NC_TEA	通用机床数据
SLQMD_ACC_CH_TEA	通道专用机床数据
SLQMD_ACC_AX_TEA	轴专用机床数据

表 11-5：有效 AccTypen - 设定数据

设定数据	描述
SLQMD_ACC_NC_SEA	一般设定数据
SLQMD_ACC_CH_SEA	通道专用设定数据
SLQMD_ACC_AX_SEA	轴专用设定数据

表 11-6：有效的 AccTypen - 用户数据

用户数据(GUD)	描述
SLQMD_ACC_CH_GUD	通道专用的 GUD
SLQMD_ACC_NC_GUD	全局 GUD

表 11-7：有效的 AccTypen - 程序特有的用户数据

程序特有的用户数据	描述
SLQMD_ACC_CH_PUD	程序特有的全局用户数据
SLQMD_ACC_CH_LUD	程序特有的本地用户数据

变量路径

机床数据和设定数据的路径可以查看标准操作区“调试”下的菜单项“机床数据”。 此处显示的名称可被直接采用，用“$”开头，没有其他前缀。

GUD 路径相当于 GUD 的名称。

驱动数据的路径不用字母开头。

表 11-8：变量路径示例

变量路径	描述
$MN_AXCONF_MACHAX_NAME_TAB[0]	访问第一个机床轴名称
$MC_CHAN_NAME	单独访问，访问一个通道名称
MY_UGUD	单独访问，访问一个自定义的 GUD
$MN_AXCONF_MACHAX_NAME_TAB[0,2]	数组访问，访问前 3 个通道轴名称
MYARRAY_UGUD[0,2]	单独访问，访问一个自定义的数组 GUD
2	单独访问，访问驱动数据 p2

11.3.2 读/写 NC 变量

同步读取一个 NC 变量

同步读取一个机床数据、设定数据或 GUD 数据（变量）。

表 11-9：readNcData

```
long SIQMd::readNcData (enum SIQMdAccEnum accType,
                        long lBtssUnit,
                        const QString& szName,
                        QString& rszValue,
                        long lGudNr = 0);

long SIQMd::readNcData (enum SIQMdAccEnum accType,
                        long lBtssUnit,
                        const QString& szName,
                        QVariant& rvValue,
                        long lGudNr = 0);
```

参数	含义
accType	AccType (见章节 11.3.1“定义”的段落“AccTypen”)
lBtssUnit	和 AccType 相关的 Btss 单位： 通道专用数据 ➔通道号 轴专用数据 ➔轴号 其他数据 ➔始终为 0
szName	变量路径 (见章节 11.3.1“定义”的段落“变量路径”)
rszValue	返回值，QString 格式（执行出错时返回“#”）。 在数组访问中，各个值用“|”隔开。
rvValue	返回值，QVariant 格式（执行出错时该 QVariant 无效::isValid() ➔ false） 此处最好检查访问类型。在数组访问中，返回值也可能是 QVariant::ByteArray，取决于变量路径 QVariant::List。
lGudNr	可选参数。 只有 AccType 是 GUD 用户数据时，才需要使用该参数。此时要为参数指定 GUD 号(1..9)。
返回值	读取任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.2“分步示例”

同步读取多个 NC 变量

同步读取多个机床数据、设定数据或 GUD 数据（变量）。
这些变量必须位于相同的区域内，具有相同的 BTSS 单位和 GUD 号。

表 11-10：readNcData

long SIQMd::readNcData (enum SIQMdAccEnum accType, long lBtssUnit, const QStringList& szNames, QVector<QVariant>& rvValues, long lGudNr = 0);	
参数	含义
accType	AccType (见章节 11.3.1“定义”的段落“AccTypen”)
lBtssUnit	和 AccType 相关的 Btss 单位： 通道专用数据 ➔通道号 轴专用数据 ➔轴号 其他数据 ➔始终为 0
szNames	多个变量路径的列表 (见章节 11.3.1“定义”的段落“变量路径”)
rvValues	返回值列表，QVariant 格式。 该列表的大小和 szNames 的列表相同。 此处最好检查访问类型。在数组访问中，返回值也可能是 QVariant::ByteArray，取决于变量路径 QVariant::List。 函数返回错误时，可以通过 QVariant::isValid 查看各个变量访问的执行状态。
lGudNr	可选参数。 只有 AccType 是 GUD 用户数据时，才需要使用该参数。此时要为参数指定 GUD 号(1..9)。
返回值	读取任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.3“分步示例”

同步写入一个 NC 变量

同步写入一个机床变量、设定变量或 GUD 变量。

表 11-11：writeNcData

long SIQMd::writeNcData(enum SIQMdAccEnum accType, long lBtssUnit, const QString& szName, const QString& szValue, long lGudNr = 0);	
参数	含义
accType	AccType (见章节 11.3.1“定义”的段落“AccTypen”)
lBtssUnit	和 AccType 相关的 Btss 单位： 通道专用数据 ➔通道号

long SIQMd::writeNcData(enum SIQMdAccEnum accType, long lBtssUnit, const QString& szName, const QString& szValue, long lGudNr = 0);	
参数	含义
	轴专用数据 ➔轴号 其他数据 ➔始终为 0
szName	变量路径 (见章节 11.3.1“定义”的段落“变量路径”)
szValue	待写入的值 在数组访问中，各个值用“|”隔开，比如[0,2] ➔ “0|1|2”。
lGudNr	可选参数。 只有 AccType 是 GUD 用户数据时，才需要使用该参数。此时要为参数指定 GUD 号(1..9)。
返回值	写入任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.2“分步示例”

同步写入多个 NC 变量

同步写入多个机床数据、设定数据或 GUD 数据（变量）。
这些变量必须位于相同的区域内，具有相同的 BTSS 单位和 GUD 号。

表 11-12：writeNcData

long SIQMd::writeNcData(enum SIQMdAccEnum accType, long lBtssUnit, const QStringList& szNames, const QStringList& szValues, long lGudNr = 0);	
参数	含义
accType	AccType (见章节 11.3.1“定义”的段落“AccTypen”)
lBtssUnit	和 AccType 相关的 Btss 单位： 通道专用数据 ➔通道号 轴专用数据 ➔轴号 其他数据 ➔始终为 0
szNames	多个变量路径的列表 (见章节 11.3.1“定义”的段落“变量路径”)
szValues	待写入值的列表。 在数组访问中，各个值用“|”隔开，比如[0,2] ➔ “0|1|2”。
lGudNr	可选参数。 只有 AccType 是 GUD 用户数据时，才需要使用该参数。此时要为参数指定 GUD 号(1..9)。
返回值	写入任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.3“分步示例”

11.3.3 读/写驱动数据

查询驱动数据配置

返回一张列出了属于某个驱动类型的所有驱动对象的列表。

表 11-13：getDrvObjectList

long SIQMd::getDrvObjectList(enum SIQMdDrvObjTypeEnum lDrvObjectType, QVector<SIQMdDrvObject>& rvDrvObjects);	
参数	含义
lDrvObjectType	支持的驱动类型： SLQMD_CFG_OBJ_TYPE_CU ➔控制单元 SLQMD_CFG_OBJ_TYPE_COM ➔通讯 SLQMD_CFG_OBJ_TYPE_DC ➔驱动控制 SLQMD_CFG_OBJ_TYPE_LM ➔电源模块 SLQMD_CFG_OBJ_TYPE_IO_COMP ➔I/O 设备
rvDrvObjects	驱动对象的列表。 （参见章节 11.4“SIQMdDrvObject 引用”）
返回值	任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.4 / 11.2.5“分步示例”

查询驱动的句柄

返回机床轴对应的驱动的句柄。在后续访问中需要使用该句柄。

表 11-14：getDrvHandle

long SIQMd::getDrvHandle(long lAxisNumber);	
参数	含义
lAxisNumber	机床轴的编号
返回值	驱动的句柄 (>0 ➔ 句柄，=0 ➔ 错误)

另一种方法是通过总线地址、从站地址和驱动对象 ID(DoId)来查询驱动的句柄。

表 11-15：getDrvHandle

long SIQMd::getDrvHandle(long lBusAddr, long lSlaveAddr, long lDoId);	
参数	含义
lBusAddr	驱动的总线地址
lSlaveAddr	驱动的从站地址
lDoId	驱动对象的 ID
返回值	驱动的句柄 (>0 ➔ 句柄，=0 ➔ 错误)

示例：
另见章节 11.2.4 / 11.2.5“分步示例”

同步读取一个驱动数据

同步读取一个驱动数据。

表 11-16：readDrvData

long SIQMd::readDrvData(long lDrvHandle, const QString& szName, QString& rszValue); long SIQMd::readDrvData(long lDrvHandle, const QString& szName, QVariant& rvValue);	
参数	含义
lDrvHandle	驱动句柄 （见段落“查询驱动的句柄”）
szName	变量路径 (见章节 11.3.1“定义”的段落“变量路径”)
rszValue	返回值，QString 格式（执行出错时返回“#”）。 在数组访问中，各个值用“|”隔开。
rvValue	返回值，QVariant 格式（执行出错时该 QVariant 无效::isValid() ➔ false） 此处最好检查访问类型。在数组访问中，返回值也可能是 QVariant::ByteArray，取决于变量路径 QVariant::List。
返回值	读取任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.4 / 11.2.5“分步示例”

同步读取多个驱动数据

同步读取多个驱动数据。这些数据必须具有相同的驱动句柄。

表 11-17：readDrvData

long SIQMd::readDrvData(long lDrvHandle, const QStringList& szNames, QVector<QVariant>& rvValues);	
参数	含义
lDrvHandle	驱动句柄 （见段落“查询驱动的句柄”）
szNames	多个变量路径的列表 (见章节 11.3.1“定义”的段落“变量路径”)
rvValues	返回值列表，QVariant 格式。 该列表的大小和 szNames 的列表相同。 此处最好检查访问类型。在数组访问中，返回值也可能是 QVariant::ByteArray，取决于变量路径 QVariant::List。 函数返回错误时，可以通过 QVariant::isValid 查看各个变量访问的执行状态。
返回值	读取任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

同步写入一个驱动数据

同步写入一个驱动数据。

表 11-18：writeDrvData

long SIQMd::writeDrvData(long lDrvHandle, const QString& szName, const QString& szValue);	
参数	含义
lDrvHandle	驱动句柄 （见段落“查询驱动的句柄”）
szName	变量路径 (见章节 11.3.1“定义”的段落“变量路径”)
szValue	待写入的值 在数组访问中，各个值用“|”隔开，比如[0,2] ➔ “0|1|2”。
返回值	写入任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.4 / 11.2.5“分步示例”

同步写入多个驱动数据

同步写入多个驱动数据。这些数据必须具有相同的驱动句柄。

表 11-19：writeDrvData

long SIQMd::writeDrvData(long lDrvHandle, const QStringList& szNames, const QStringList& szValues);	
参数	含义
lDrvHandle	驱动句柄 （见段落“查询驱动的句柄”）
szNames	多个变量路径的列表 (见章节 11.3.1“定义”的段落“变量路径”)
szValues	待写入值的列表。 在数组访问中，各个值用“|”隔开，比如[0,2] ➔ “0|1|2”。
返回值	写入任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

11.3.4 查询用于 CAP 服务的变量名称

用于 CAP 服务的 NC 变量名称

查询用于 CAP 服务的一个或多个 NC 变量的名称，NC 变量包括机床数据、设定数据和 GUD 数据。这些变量必须位于相同的区域内，具有相同的 BTSS 单位和 GUD 号。

表 11-20：getNcDataLinkItem

long SIQMd::getNcDataLinkItem(enum SIQMdAccEnum accType, long lBtssUnit, const QStringList& szNames, QStringList& rszLinkItems, long lGudNr = 0);	
参数	**含义**
accType	AccType (见章节 11.3.1“定义”的段落“AccTypen”)
lBtssUnit	和 AccType 相关的 Btss 单位： 通道专用数据 ➔通道号 轴专用数据 ➔轴号 其他数据 ➔始终为 0
szNames	多个变量路径的列表 (见章节 11.3.1“定义”的段落“变量路径”)
rszLinkItems	一张列出返回的用于 CAP 服务的 LinkItem 的列表。 该列表的大小和 szNames 的列表相同。
lGudNr	可选参数。 只有 AccType 是 GUD 用户数据时，才需要使用该参数。此时要为参数指定 GUD 号(1..9)。
返回值	任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

示例：
另见章节 11.2.2 / 11.2.3“分步示例”

11.3.5 其他函数

执行 PI 服务_N_CONFIG

如果修改了生效级“NEW_CONF”的机床数据，则需要执行 PI 服务_N_CONFIG 来激活该数据。

表 11-21：piConfig

long SIQMd::piConfig();	
参数	**含义**
返回值	任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

备份驱动对象

在闪存卡上备份驱动对象。此时，驱动对象由驱动句柄标识。

表 11-22：drvSave

long SIQMd::drvSave(enum SIQMdSnxActionEnum lActionType, long lDrvHandle, SIQMdTextInfoCb* pCallback = 0);	
参数	**含义**
lActionType	允许使用下列类型： SLQMD_SNXACT_AKTDO_ONLY ➔利用 p971=1 备份当前驱动对象 SLQMD_SNXACT_AKTCU_ONLY ➔设置 p977=1，备份驱动句柄标出的驱动对象“控制单元” SLQMD_SNXACT_ALLCU

long SIQMd::drvSave(enum SIQMdSnxActionEnum IActionType, long IDrvHandle, SIQMdTextInfoCb* pCallback = 0);	
参数	含义
	➔设置 p977=1，备份所有控制单元
IDrvHandle	需要备份的驱动的句柄。 （见段落 11.3.3“读/写驱动数据”的段落“查询驱动的句柄”） 此处传送 0 时备份所有控制单元。
pCallback	无含义（不传送任何内容）
返回值	任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

该函数发送以下信号：onSIQMdTextInfo。

执行重启

对由驱动句柄标出的控制单元执行复位。必要时还要一同执行一次 NCK 复位。

表 11-23：复位

long SIQMd::reset(enum SIQMdSnxActionEnum IActionType, long IDrvHandle, SIQMdTextInfoCb* pCallback = 0);	
参数	含义
IActionType	允许使用下列类型： SLQMD_SNXACT_AKTCU_ONLY ➔设置 p976=2，复位由驱动句柄标出的控制单元 SLQMD_SNXACT_ALLCU ➔设置 p976=2，复位所有控制单元 SLQMD_SNXACT_NCK_ONLY ➔仅 NCK 复位 SLQMD_SNXACT_NCK_AND_AKTCU ➔设置 p976=2，复位由驱动句柄标出的控制单元和 NCK SLQMD_SNXACT_NCK_AND_ALLCU ➔设置 p976=2，复位所有控制单元和 NCK SLQMD_SNXACT_ALLCU_PREPARE ➔设置 p972=3，准备好复位所有控制单元。在进行下一次 NCK 复位后，会复位所有控制单元。
IDrvHandle	控制单元的驱动句柄 （见段落 11.3.3“读/写驱动数据”的段落“查询驱动的句柄”） 在以下“IActionType”类型中此处给出 0： SLQMD_SNXACT_ALLCU SLQMD_SNXACT_NCK_ONLY SLQMD_SNXACT_NCK_AND_ALLCU SLQMD_SNXACT_ALLCU_PREPARE
pCallback	无含义（不传送任何内容）
返回值	任务的执行状态（0 ➔ 正常， <>0 ➔ 出错）

该函数发送以下信号：onSIQMdTextInfo。

检查，是否所有驱动都可访问

该函数可用于检查，是否所有已配置的驱动都可访问。

表 11-24：piConfig

bool SIQMd::areAllDrivesAccessible();	
参数	含义
返回值	true ➔所有已配置的驱动都可访问 false ➔至少有一个已配置的驱动无法访问

11.3.6 通知/信号

概述

上文 11.3 说明的函数会发送一些信号。这些信号在下文详细说明。

onSIQMdTextInfo

该信号提供 `drvSave` 和 `reset` 任务的信息。

表 11-25：onSIQMdTextInfo

void onSIQMdTextInfo(SIQMdDlgTextEnum drvInfoTextType, QString szPar1, QString szPar2);	
参数	含义
drvInfoTextType	允许的类型有： SLQMDTXT_TEXT_ONLY ➔一条文本信息。文本无需翻译。 对应的文本在参数“szPar1”中。 SLQMDTXT_SAVE ➔驱动对象名称保存在参数“szPar1”中。 参数“szPar2”的值域为 0%...100%。
szPar1	参见“drvInfoTextType”
szPar2	参见“drvInfoTextType”

onSIQMdAllDrivesAccessible

该信号报告所有已配置的驱动现在都可访问。

表 11-26：onSIQMdTextInfo

void onSIQMdAllDrivesAccessible ();

11.4 SlQMdDrvObject 引用

11.4.1 定义

概述

用于保存驱动数据配置的数据结构。

表 11-27：数据结构 SlQMdDrvObject

日期	含义
long m_lDrvHandle;	驱动对象的句柄
long m_lBusAddr;	驱动对象的总线地址
long m_lSlaveAddr;	驱动对象的从站地址
long m_lDoId;	驱动对象 ID
long m_lDoTypeId;	控制单元的参数 p107
long m_lAxNr;	指定的轴号，没有指定轴时为 0
QString m_szDoName;	驱动对象的名称
QString m_szAxName;	指定轴的名称

I 索引

I.1 关键词索引