

野人家园

UartAssist串口调试助手

软件操作手册

Ver 5.0.8

文档最后更新：2023/08/01

技术支持：support@cmssoft.cn

软件授权：南京云想物联网科技有限公司

软件下载 <http://www.cmssoft.cn/download/cmssoft/uartassist.zip>

目录

第一章 UartAssist 简介	4
1.1 软件特色	4
1.2 运行环境	5
1.3 软件安装	5
1.4 应用场景	5
1.5 软件界面	6
第二章 串口通信原理	10
2.1 串口划分标准	10
2.2 串口通信协议	11
2.3 UART-TTL	11
2.4 UATT-RS232	12
2.5 UART-RS485	12
2.6 UART-RS422	13
第三章 调试助手配置选项	14
3.1 语言及编码	14
3.2 窗口主题样式	15
3.3 数据接收设置选项	17
3.4 数据发送设置选项	20
3.5 发送框默认内容设置	23
3.6 其它参数及控制选项	23
第四章 调试助手基本操作	25
4.1 串口参数设置	25
4.2 打开串口连接	26
4.3 串口数据接收	26
4.4 串口数据发送	27
4.5 发送转义字符	28
4.6 发送指令脚本	28
第五章 调试助手进阶选项	31
5.1 快捷指令	31
5.2 批量发送	33
5.3 自动应答	34
5.4 历史发送	34
5.5 校验计算器	35
5.6 ASCII 码对照表	36
5.7 命令行启动参数	37
第六章 脚本代码语法规则	38

6.1 运算符	38
6.2 运算表达式	38
6.3 BLOCK 代码块	39
6.4 变量数据类型	39
6.5 变量定义及作用域	40
6.6 变量强制类型转换	40
6.7 语法大小写规则	41
6.8 字段注解的定义及引用	41
6.9 内建系统函数详解	42
第七章 自动应答规则设计	50
7.1 应答规则概述	50
7.2 应答规则入门	51
7.3 指令匹配模板	54
7.4 指令应答模板	58
7.5 应答规则设计实例	60

第一章 UartAssist 简介

野人家园UartAssist串口调试助手软件，是Windows平台下的串口通信调试工具，广泛应用于工控领域的数据监控、数据采集、数据分析等工作，是串口应用开发及调试工作必备的专业工具之一，可以帮串口通信项目的应用设计、开发、测试人员检查所开发的串口通信应用软/硬件产品的数据收发状况，提高开发速度，简化开发复杂度，是串口通信应用开发调试的得力助手。UartAssist串口调试助手是绿色软件，无需安装，只有一个执行文件，适用于各版本Windows操作系统，不需要微软dotNet框架支持。可以在一台PC上同时启动多个串口调试助手（使用不同的COM口）。典型应用场合：通过串口调试助手与自行开发的串口程序或者串口设备进行通信联调。支持多串口，自动监测枚举本地可用串口；自由设置串口号、波特率、校验位、数据位和停止位等（支持自定义非标准波特率）；支持对串口DCD、DTR、DSR、RTS等针脚状态位的检测控制。支持ASCII/Hex两种模式的数据收发，发送和接收的数据可以在16进制和AscII码之间任意转换；可以自动发送校验位，支持多种校验格式；支持间隔发送，循环发送，批处理发送，输入数据可以从外部文件导入；可以保存预定义指令/数据序列，任何时候都可以通过工具面板发送预定义的指令或数据，或进行批量指令发送，便于通信联调。软件界面支持中/英文（通过菜单选项选择切换），默认自适应操作系统的语言环境。

1.1 软件特色

- ◆ 绿色软件、只有一个执行文件、无需安装；
- ◆ 支持中英文双语言，自动根据操作系统环境选择系统语言类型；
- ◆ 支持常用的各种波特率，端口号、校验位、数据位和停止位均可设置；
- ◆ 支持软/硬件多种流控方式；
- ◆ 自动检测枚举本机串口号，支持虚拟串口；
- ◆ 支持设置分包参数（最大包长、分包时间），防止接收时数据粘包。
- ◆ 支持ASCII/HEX码数据发送,发送和接收的数据可以在十六进制码和ASCII码之间任意转换，支持发送和显示汉字；
- ◆ 可以自动发送校验位，支持多种校验格式，如校验和、LRC、BCC、CRC8、CRC16、CRC32、MD5等，其中CRC校验码可任意定制CRC参数(CRC多项式、初始值、输入反转、输出反转、输出异或值)；
- ◆ 发送内容支持转义字符。例如，发送框中文本包含诸如\r\n等转义符时，会自动解析成对应的ASCII码进行发送。
- ◆ 支持AT指令自动添加回车换行选项，启用该选项时，在发送AT指定时会自动在行尾补全回车换行符；
- ◆ 可以通过输入框发送数据，也可以从文件数据源发送数据；
- ◆ 支持接收数据自动保存到文件，并且文件类型支持数据文件和日志文件两种格式，其中数据文件只保存接收的数据内容，而日志文件则会保存调试助手完整的数据收发日志信息。

- ◆ 支持日志接收模式：启用该选项后在接收窗口显示接收内容时自动显示时间戳等相关信息。
- ◆ 支持任意间隔发送，循环发送；
- ◆ 接收和发送的文字编码支持ANSI (GBK) 与UTF8两种方式，并且接收编码与发送编码可以独立设置，互不影响；
- ◆ 支持预定义指令/数据，可通过按键或者自定义快捷键发送预定义指令，预定义指令/数据列表可以按文件的方式保存、导入和导出；
- ◆ 支持批量发送指令/数据序列，可设置每条指令的发送延迟，并可按设定顺序及延迟时间依次批量发送。批量定义的数据/指令可以保存、导入和导出。
- ◆ 自动保存历史发送记录，可以通过历史记录发送历史数据；
- ◆ 支持界面窗口的字体以及背景定制；
- ◆ 支持工作界面精简模式（主界面左侧面板可折叠收起）；
- ◆ 可定制发送框默认数据内容。

1.2 运行环境

软件运行环境为Windows平台, 包括Windows95/WinXP/Vista/Win7/Win8/Win10/WinALL, 兼容32位/64位操作系统。

1.3 软件安装

绿色软件, 解压后只有一个执行文件, 直接运行即可。无需安装(不依赖)Microsoft .NET Framework框架。

1.4 应用场景

串口调试助手通过对PC端口串口(COM口或USB串口)的读写操作, 实现对串口设备或者串口应用程序的通信联调。通过串口数据的抓取、记录、分析以及数据/指令的发送控制, 实现对目标串口设备或者串口应用程序的通信能力以及通信行为的分析、验证。总的来说, 串口调试助手, 主要有以下几类应用场景。

(1) 串口终端(仪器设备)的参数设置。工程应用中为了方便终端设备的参数设置, 可通过串口调试助手建立到串口终端设备的串口连接, 然后直接在串口调试助手对串设备进行参数设置。

(2) 串口终端(仪器设备)的操作控制、串口数据的抓取、记录及分析。在工程应用中, 某些场景下需要对串口设备发送指令从而实现对设备的控制操作, 或者需要对串口设备的数据进行抓包记录。通过串口调试助手可以定时向串口终端发送指令数据, 并自动将接收及发送的报文数据, 并按日志的形式保存至磁盘文件, 以便于用户对设备的状态数据进行分析统计。

(3) 工控设备/单片机的开发调试。在单片机/嵌入式系统的串口开发过程中, 可通过串口调试助手接收单片机设备的串口数据, 或者向单片机设备发送串口数据, 配合单片机程序开发, 验证单片机程序的通信能力以及业务逻辑的准确性; 或者通过串口调试助手对单片机设备进行数据疲劳测试(通过批量或者循环指令发送), 并记录通信过程中的数据交互日志,

实现串口产品在研发过程中的可靠性验证。

1.5 软件界面

UartAssist串口调试助手的主要功能界面如图1-1~1-7所示，包括主界面及工具面板窗口各项功能构成，而具体的功能描述则会在后面章节中具体介绍。

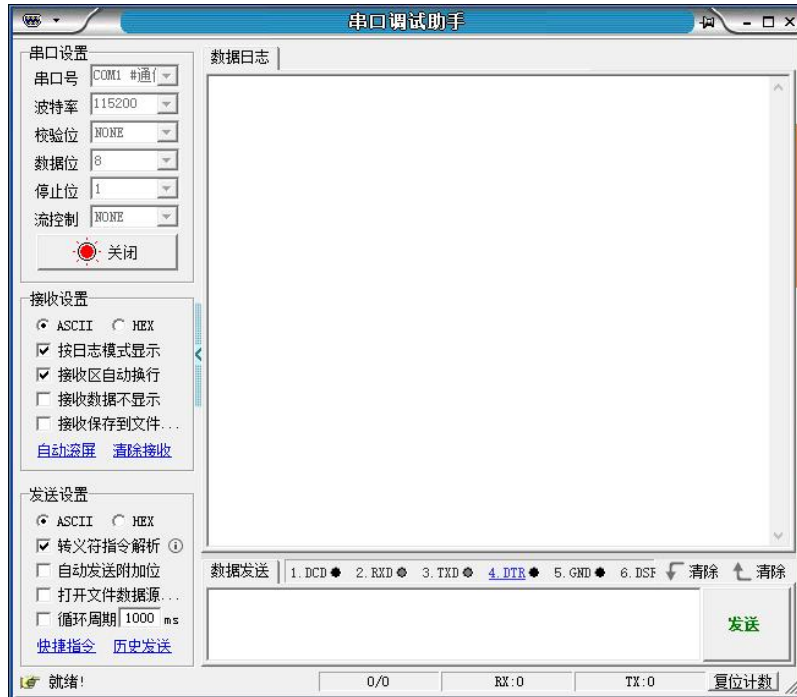


图1-1 软件主界面

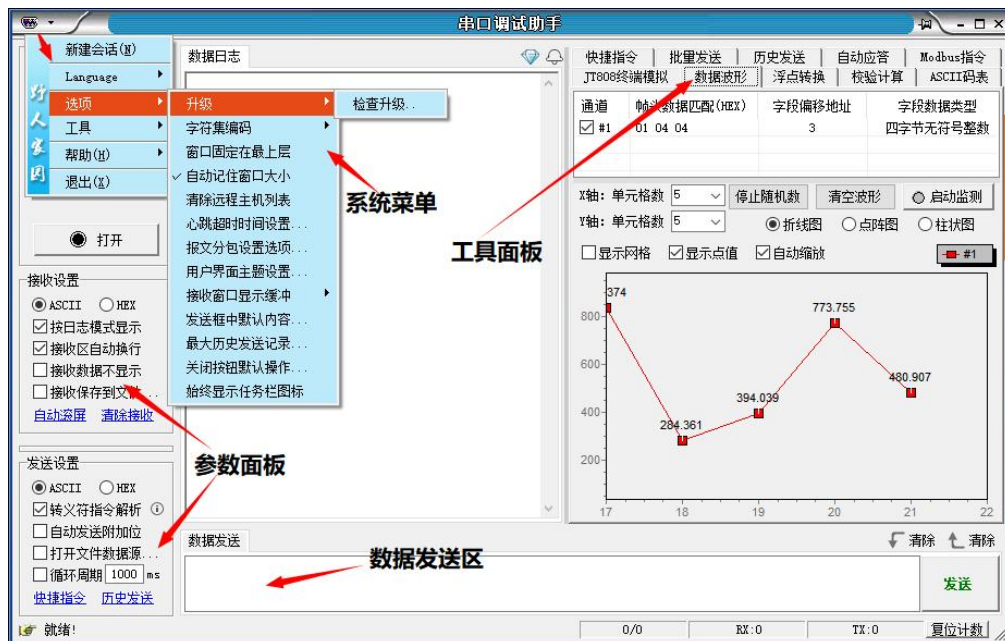


图1-2 界面基本构成

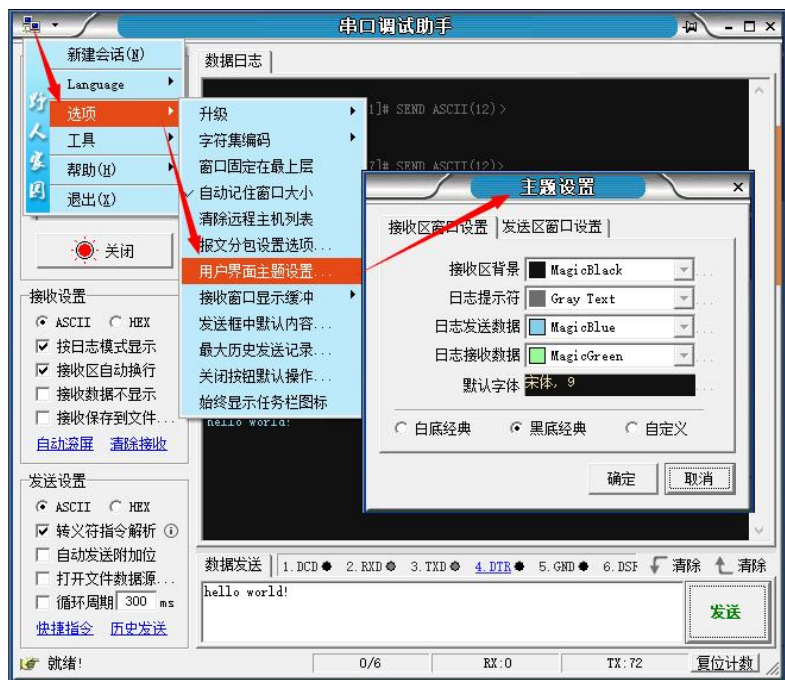


图1-3 界面主题（背景/字体）设置



图1-4 工具面板—快捷指令



图1-5 工具面板/批量发送



图1-6 工具面板/自动应答



图1-7 工具面板/历史发送

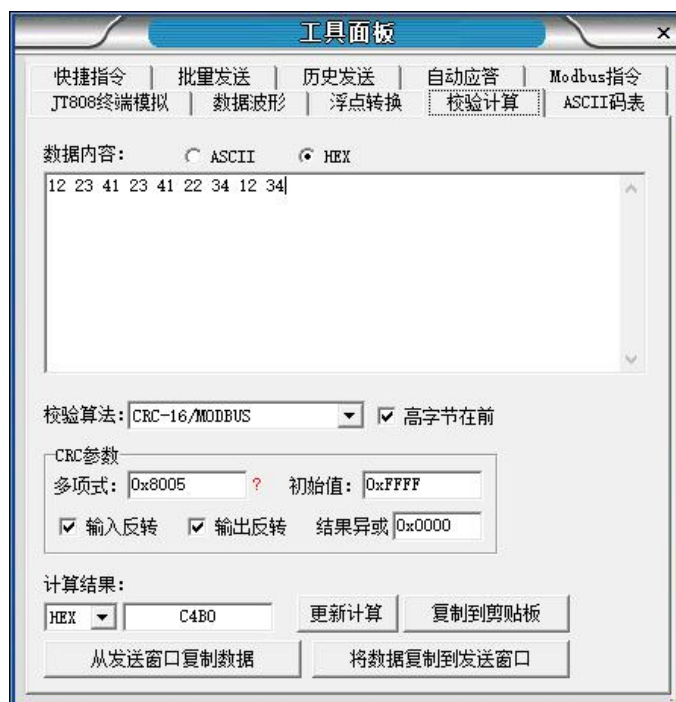


图1-8 工具面板/检验计算器

第二章 串口通信原理

串行接口简称串口，也称串行通信接口（通常指COM接口），是采用串行通信方式的扩展接口。串行通讯的特点是：数据是按位(bit)逐位依次传输的，只需一根传输线即可完成单向传输通信；如果有一对传输线就可以实现双向全双工通信（可以直接利用电话线作为传输线），从而大大降低了成本，特别适用于远距离通信，但传送速度较慢。

2.1 串口划分标准

串行接口按时钟同步方式不同，可以分为同步串行接口与异步串行接口。

同步串行通信接口（Synchronous Serial Interface，简称SSI）是一种常用的工业用通信接口。常见的SPI接口也是同步串行接口的一种。

异步串行通信接口，一般特指通用异步收发器（Universal Asynchronous Receiver/Transmitter，简称UART）。它包括了RS232、RS499、RS423、RS422和RS485等接口标准规范和总线标准规范，即UART是异步串行通信口的总称。说到串口，如果不指名同步或异步，则通常特指异步串行接口。

UART本身并没有规定电气特性，而RS232、RS499、RS423、RS422和RS485等，是对应各种异步串行通信口的接口标准和总线标准，它规定了通信口的电气特性、传输速率、连接特性和接口的机械特性等内容。实际上是属于通信网络中的物理层（最底层）的概念，与通信协议没有直接关系。而通信协议，是属于通信网络中的数据链路层（上一层）的概念。

（1）UART-TTL电平标准：TXD/RXD/DTR/CTS信号的逻辑“0”输出电平要求小于0.4V；逻辑“1”输出电平要求大于2.4V，典型3.4V，上限不超过TTL电源5V。

（2）UART-RS232标准：RS（Recommended Standard）代表推荐标准，232是标识号。RS232标准主要规定了指电平定义、信号引脚功能的定义及应答协议等。RS232是全双工的，最小只需要3根信号线，一根数据线专门负责发送数据，另一根数据线专门接收数据，还有一根共地线。但是RS232只能单机通信。RS232接口标准定义的电平是负逻辑而且电压范围大：“1”（-3V~-15V），“0”（+3~+15V），所以，当一个微控制器中的UART相连于PC时，它需要一个RS232驱动器来转换电平。RS232电平是UART的默认标准，除此之外还有TTL电平以及CMOS电平，工程应用中要注意匹配。通常DIY人士常说的TTL线刷工具，就是将设备的TTL_UART口转接到PC的USB口的转换连接器。

（3）UART-RS485标准：RS485是采用差分线来进行串行数据通信的总线型接口规范。在要求通信距离为几十米到上千米时，广泛采用RS-485 串行总线标准。RS-485采用平衡发送和差分接收，因此具有抑制共模干扰的能力。加上总线收发器具有高灵敏度，能检测低至200mV的电压，故传输信号能在千米以外得到恢复。RS485采用差分信号负逻辑，+2V~+6V表示“0”，-6V~-2V表示“1”。RS485有两线制和四线制两种接线，四线制只能实现点对点的通信方式，现很少采用，现在多采用的是两线制接线方式，这种接线方式为总线式拓扑结构在同一总线上最多可以挂接32个结点。

（4）UART-RS485标准：实际上就是具有两对差分线的RS485，可实现全双工通信。

2.2 串口通信协议

UART协议是全双工的异步收发协议,数据发送信号线TX与接收信号线RX各自独立并行工作,互不影响。通讯双方的传输速度(波特率)、检验位类型、停止位数目需要事先相互约定,否则无法正确通讯。通过UART协议每传输的一个字节数据,都要在前面增加其同步作为的起始位,后面还要增加校验位及停止位。起始位是逻辑“0”电平,要求必须从下降沿开始,也就是说在起始位前沿必须是“1”(停止位或者空闲位);而停止位没有前沿的要求,只要是“1”即可。在不传输数据时,数据线上必须保持“1”(空闲位)。注意,除了空闲位,其他位的宽度都是相同的(等于波特率的)。如果串口数据连续无延迟发送,则每一字节发送完毕后经过停止位后不会插入空闲位。

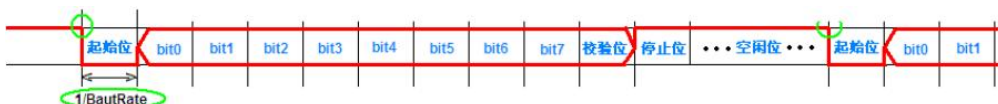


图 2-1 UART 数据传输时序

关于波特率的概念说明:波特率(BaudRate)表示每秒钟传输的bit数,其单位是bps(bit per second),这里的bit不仅仅是指数据位,还包括起始位、校验位、停止位,而且每1位的时间宽度都是是 $1/\text{BaudRate}$ 。假设串口配置为(9600bps、1个起始位、8个数据位、1个奇校验位、1个停止位,共11位),则每秒可以传输的字节数为 $9600/11=872$ Bytes。

2.3 UART-TTL

UART-TTL就是采用TTL电平的全双工UART接口。也就是说,UART串口的各I/O引脚,包括TXD、RXD、DTR、CTS等信号逻辑“0”输出电平要求小于0.4V;逻辑“1”输出电平要求大于2.4V,典型3.4V,上限不超过TTL电源5V。

UART-TTL接口的常见实物形式有: USB转TTL串口线(刷机线/下载线)、USB转TTL串口适配器(USB Dongle),如下图所示。



图 2-2 UART-TTL 适配器

由于UART-TTL的应用场景中,通常只需要RXD/TXD/GND/VCC这4根线,因此常见的USB转TTL接口也只引出这四根线。

2.4 UATT-RS232

UART-RS232就是采用232电平的全双工UART接口。RS232接口标准定义的电平是负逻辑而且电压范围大：“1”（-3V~-15V），“0”（+3~+15V）。RS232电平是UART的默认标准，由于RS232电平幅值较大，相比TTL电平具更大的噪声容限及更长的传输距离。RS232串口最大传输距离标准值为15米, 最长距离不要超过20米。

UATT-RS232的接口形式，通常都是DB9形式，即9针D形接口，并且分公头母头两种形式，如下图所示：



图 2-3 UART-RS232-DB9 接口公母头

Pin No.	Designation Description	Input/Output
1 DCD	Data carrier detect	Input
2 RxD	Receive data	Input
3 TxD	Transmit data	Output
4 DTR	Data terminal ready	Output
5 GND	Ground –	
6 DSR	Data set ready	Input
7 RTS	Request to send	Output
8 CTS	Clear to send	Input
9 RI	Incoming call	Input

图 2-4 UART-RS232 串口引脚定义 (DB9)

2.5 UART-RS485

UART-RS485属于总线类型，最小只需要两根数据线（差分信号：A线与B线），差分线之间的压差反映传输信号值，因此可以抵制共模干扰，抗干扰能力强，理论传输距离可以达到1200米。

RS485电平的逻辑1（正）是指电压B>A的情形，逻辑0（负）则是指电压A>B的情形。并且对于RS485电平发送器，发送的差分电平要求满足A线与B线间电压差的绝对值在2v~6v之间；对于RS485电平接收器，只要求检测到AB线之间的压差绝对值不小于200mv即可表示数据，否则表示总线空闲。单组差分信号线的RS485设备只能半双工通信，也就是同一时刻要么接收数据，要么发送数据（设备的输入/输出模式的切换有专门的引脚控制）。由于RS485没有

总线仲裁机制，只能有上层软件来保证通信可靠性。因此，用户编程时必须遵循二个原则：

（a）“仅在要发送数据时才强占总线（将设备切换为发送模式），一旦发送完数据则立即释放总线（切换为输入模式）”；（b）强占总线前最好先检测下总线是否空闲（比如，用低电平或高电平保持一定的时间作为空闲状态）。RS485与PC通信时，通常将RS485转换为RS232。

为了抑制干扰，RS-485需要2个终接电阻，接在传输总线的两端，其阻值要求等于传输电缆的特性阻抗，通常取120欧的电阻。在短距离传输时可不需终接电阻，即一般在300米以下不需终接电阻。

2.6 UART-RS422

RS422和RS485都是串行总线类型，电气特性完全一样。唯一的区别是RS422有两对差分线，可以实现全双工通信；而RS485只有一对差分线，只能半双工通信。

第三章 调试助手配置选项

调试助手的配置参数及控制选项众多,涉及的内容包括系统语言、文字编码、窗口样式、数据格式、数据校验、转义字符支持、数据发送控制方式、数据接收存储及日志文件记录等等。使用调试助手进行通信调试时,结合实际应用场景,使用恰当的配置选项,可以有效地提高通信调试的工作效率,甚至达到事半功倍的效果。

3.1 语言及编码

调试助手软件支持中、英文双语,默认情况下自动根据系统语言选择切换。在中文环境下自动选择中文,其它语言环境自动选择为英文。也可以通过调试助手的[Language]菜单选项,直接指定中/文语言。



图 3-1 语言以及编码切换

调试助手的数据编码支持 ANSI (GBK) /UTF-8 两种编码。默认方式是 ANSI (GBK) 编码。如果接收窗口显示乱码数据,有可能接收到的数据是 UTF-8 编码的文字,如果按照默认的 ANSI (GBK) 方式显示就会乱码。解决方法是,在接收窗口中点击右键,在弹出菜单中,选择切换编码为 UTF-8,如上图 3-1 所示。注意,切换编码后,并不会刷新显示已接收的乱码数据,只有后来新接收的数据才能按新的编码设置来显示。

发送数据时,也涉及到文字编码方式的选择。发送窗口的编码方式与接收窗口的编码方式是相互独立的。同接收窗口的编码设置方式一样,在发送窗口点击右键,在弹出的右键菜单,可以选择发送数据的编码方式为 ANSI (GBK) 或 UTF-8。

对于英文字母或字符，不需要区分是 ANSI (GBK) 编码还是 UTF-8 编码，因为这两种编码是相同的，都是一个字节的 ASCII 码；但是对于汉字等多字节编码的文字或符号则是不一样的。比如，1 个汉字的 ANSI (GBK) 编码占 2 个字节，而 1 个汉字的 UTF-8 编码则占 3 个字节。调试助手能够按照所设置的编码方式对收发数据进行相应的编码及显示处理。比如，在发送窗口输入汉字“你好”，然后点击【发送】按钮时，如果按 ANSI (GBK) 编码发送，则实际发送的 16 进制数据是“C4 E3 BA C3”，而按 UTF-8 编码发送时，实际发送的 16 进制数据是“E4 BD A0 E5 A5 BD”。调试助手会按照当前设定的编码发送正确的数据。

3.2 窗口主题样式

为了确保软件界面更符合个人视觉习惯，调试助手提供了界面主题定制功能。界面主题定制的内容，包括字体样式、大小、颜色及背景。并且，日志提示符、接收的数据、发送的数据，其字体颜色可以分别独立设置。如图 3-2 所示，通过菜单选项，调出用户界面主题设置窗口。可以分别对接收窗口以及发送窗口的界面主题进行定制，或者选择使用预设的主题样式。

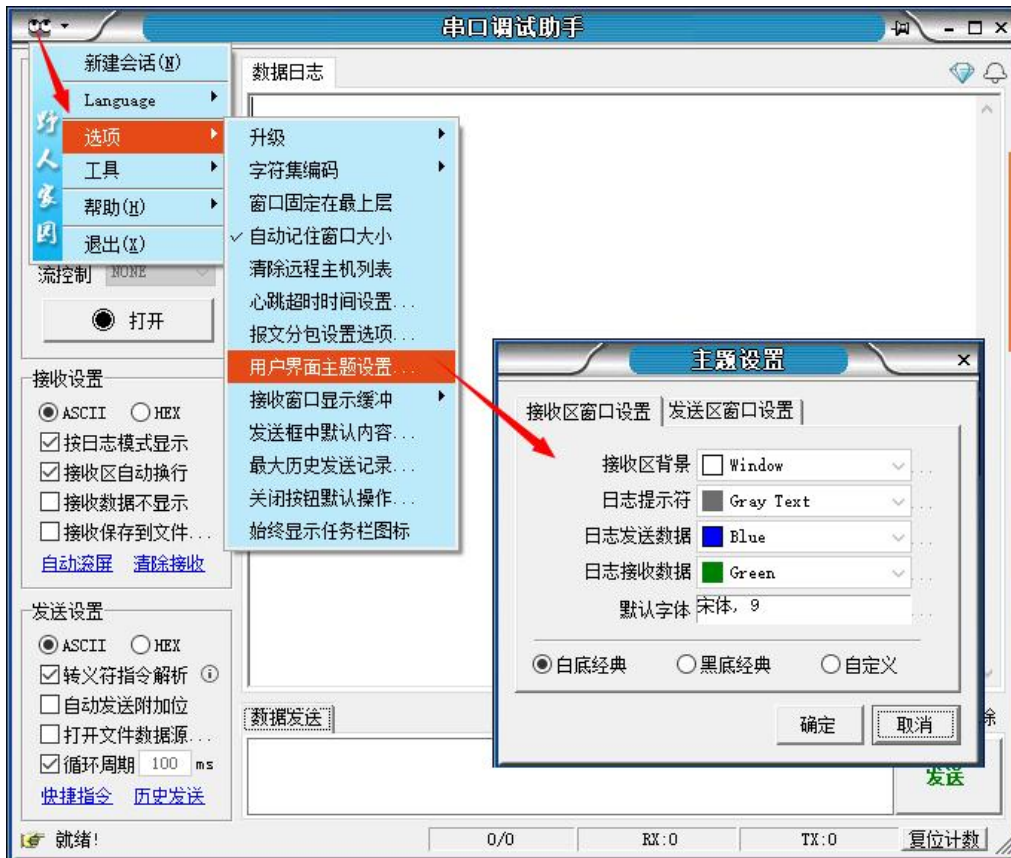


图 3-2 用户界面主题设置

调试助手提供二个预置的主题样式：白底经典（默认主题）和黑底经典。也可以选择自定义方式，并按个人的喜好习惯设置界面背景以及字体。下图是选择黑底经典主题的效果。左侧控制面板可以折叠收起，显示效果会更加简洁。

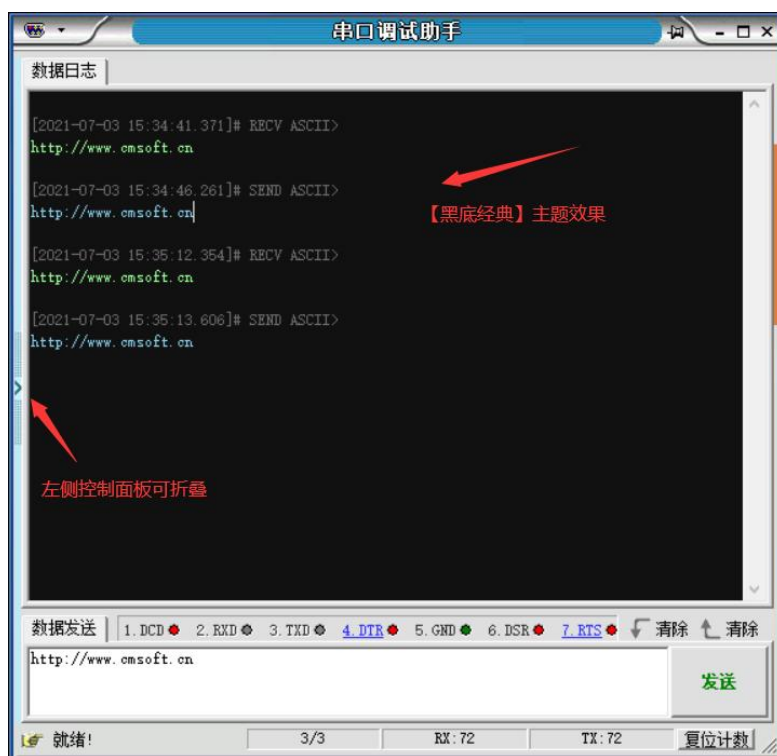


图 3-3 收/发窗口分别是“黑底经典”及“白底经典”主题模式

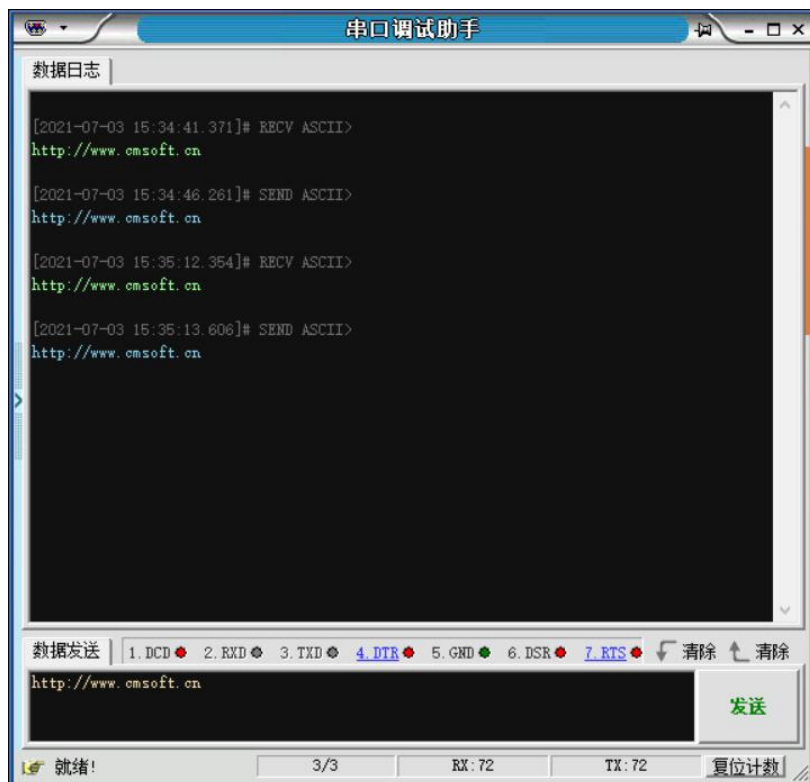


图 3-4 收/发窗口都是“黑底经典”

3.3 数据接收设置选项

3.3.1 数据接收格式

在调试助手左侧的接收参数面板，可以设置接收数据的显示格式为 ASCII 码或者 HEX 码，方便用户按不同的方式分析查看其所接收的数据。如果接收到的是可打印字符(文字)，可以直接按 ASCII 模式查看，或者按 HEX 模式查看其 16 进制编码；但如果接收到的数据包含非打印字符，那么按 ASCII 模式查看，接收窗口就有可能出现乱码数据，不能有效反映出实接收到的数据内容。这种情形下，可以选择 HEX 模式，就可以如实有效地打印出实际接收的数据内容。



图 3-5 数据接收格式控制

如上图所示，分别按 ASCII 码与 HEX 码接收一段包含非打印字符的数据。在按 ASCII 码接收时包含乱码，而按 HEX 码接收时，就是很好的分辨实际接收的内容。使用调试助手时，要根据实际的场景需求来切换数据显示格式。

3.3.2 日志显示格式

数据接收窗口默认是按日志模式显示的，除了显示接收到的数据内容外，还会显示接收数据的时间戳、数据格式(ASCII 码/HEX 码)、数据来源 IP 地址及端口号。另外，接收窗口还会显示所发送的数据记录信息。如下图所示：



图 3-6 按日志模式显示数据

如果在接收设置选项中，取消【按日志模式显示】选项。那么接收窗口就仅显示接收到的数据内容，而不会显示接收记录的时间戳等附加信息，并且发送数据记录也不会显示。

3.3.3 接收自动换行

【接收区自动换行】选项，决定每接收到一条新的数据记录是否会自动换行，还是在之前接收到的数据末尾追加。这个选项对于日志模式无明显效果，因为日志模式下每一条接收记录都会强制换行。

自动换行涉及到一个概念问题，就是如何区分一条数据记录的起始。换句话说，在接收连续的数据流时，如何将首尾相连、依次传输的多个报文切开开来，也就是处理粘包问题。由于串口数据传输类似于流式数据传输，只能保证发出的数据或指令包有序地传输，并不会给应用层提供报文分割的起始标识。即使串口发送端是一次性打包发出去的数据，到达接收端时的数据也会像流水一样陆陆续续地到达，尤其是当多个报文的发送间隔很小，并且串口应用程序由于运算能力或者算法效率问题无法及时从串口硬件接收队列取走数据时，导致应用程序无法正确地判断出原始数据包之间的时间间隙，从而无法有效的间隔密集传输的多个报文有效地分割开来。对于串口调试助手，处理数据粘包的问题比较简单粗暴，超过 50ms 的间隙就自动截断，作为一条数据记录来对待。

3.3.4 接收报存到文件

调试助手接收的数据可以自动保存到文件。在接收设置中，点击【接收保存到文件】选项，会弹出下图 3-7 所示对话框，选择接收保存的文件路径。



图 3-7 接收保存到文件

如图所示，选择目标文件存储路径时，要注意设置文件保存类型，可选项有数据文件、日志文件、滚动日志三种。所谓数据文件，就是目标文件只保存接收到原始数据内容，不包含诸如时间戳等其它附加信息，并且数据文件只包含接收数据，而不包含发送的数据记录；对于日志文件，不仅存储接收的数据内容，还存储接收数据的时间戳及数据来源等相关信息，同时还包含调试助手的发送记录信息；滚动日志则是日志文件的扩展，其每小时生成一个独立的日志文件，保存一个月，循环滚动覆盖。在实际应用中，如果纯粹是用于保存接收到的原始数据，那就选择数据文件类型；如果要记录收发双发交互的数据及发生的时间，那就选择日志文件或滚动日志类型，其中滚动日志专用于长时间的日志记录。

3.3.5 接收自动滚屏

串口调试助手接收到的数据都会显示在接收窗口，新收到的数据会自动追加到之前接收到的数据末尾，并且接收窗口会实时刷新，如果接收到的数据内容超过窗口高度时就会自动逐行滚屏，保证最新接收到的数据始终在接收窗口可见。这样就会出现一个问题，如果高速接收显示连续的数据流时，接收窗口就会不断滚屏翻页，导致用户无法有效地查看数据，因为还没看清楚就已被自动翻页。

接收设置中的【自动滚屏】选项，可以切换自动滚屏功能。软件启动后的默认设置是启用自动滚屏，单击该选项便可切换/关闭自动滚屏。一旦关闭自动滚屏，虽然新收到的数据还是会自动追加到接收窗口的数据末尾，但是接收窗口不会自动滚屏，配合鼠标控制接收窗口的滚动条，用户就可以自由查看接收窗口中的数据，不会再被自动滚屏功能所干扰。

3.4 数据发送设置选项

数据发送设置选项包括数据发送模式（ASCII/HEX）、转义字符支持、AT 指令自动回车、自动发送校验位、文件数据源设置、循环发送设置等。

3.4.1 数据发送类型

调试助手可发送的数据类型有 ASCII 文本字符串和 HEX 十六进制编码数据两种。对应软件界面左侧的发送参数面板，可以选择数据发送编码类型：ASCII 码或 HEX 码。任何数据都可以编码成十六进制形式文本进行发送，但不是任何数据都可以按 ASCII 码字符串发送。如果发送的数据包含非打印字符，那么转换为 ASCII 码就会发生乱码，发送时就会丢数据。因此，包含非打印字符的数据只能按十六进制编码进行发送；而不包含非打印字符的数据该如何选择数据发送类型，就要根据具体应用场景怎么方便就怎么选择。

3.4.2 转义字符支持

串口调试助手支持在发送的 ASCII 文本中插入转义字符。只要勾选发送设置中的【自动解析转义符】选项，发送包含转义符的 ASCII 文本时，转义符会自动解析成对应的 ASCII 码数据进行发送，方便用户以文本形式发送非打印符。转义符以反斜杠开头。比如，回车符的转义字符是\r 或\x0d、换行符的转义符是\n 或\x0a，等等。任何 ASCII 码字符（包括可打印字符及不可打印字符）都可以通过\x 后紧跟两位十六进制编码数据来表示。

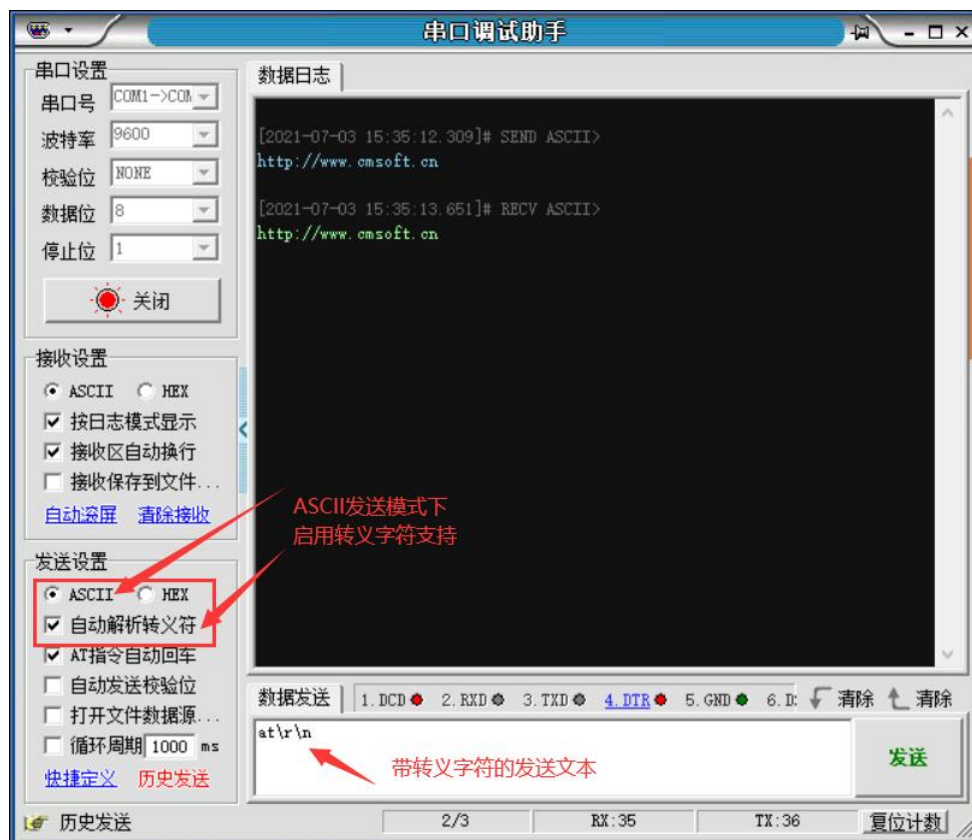


图 3-8 发送带转义符的 ASCII 码文本

上图 3-8 是一个发送文本中包含转义符的例子：在发送框中输入 hello\r\n，然后点

击发送按钮，表示发送字符串 hello 及回车换行。注意，只有在 ASCII 码发送模式下，并勾选了【自动解析转义符】选项后才支持转义字符。

3.4.3 AT 指令自动回车

为方便 AT 指令的发送，只要勾选【AT 指令自动回车】选项，那么在发送以 AT 开头文本指令时，调试助手软件会自动检查是否以回车换行符结尾，如果没有显式的回车换行结尾符，那么发送的 AT 指令自动会在尾部补齐回车换行符。这样，用户发送 AT 指令时，就不必每次都添加回车换行符，从而可有效提高通信调试的工作效率。

3.4.4 自动发送校验位

在使用调试助手进行通信调试时，某些场景下需要发送带校验位的指令数据。比如，调试 modbus 通信协议时，指令末尾需要加 CRC16 校验位。这就要求在准备调试指令前事先计算好校验码，如果遇到指令比较多且随时要修改的情形，就会比较麻烦。针对这个问题，调试助手提供了自动发送校验位功能选项。只要勾选该选项，并选择对应的校验算法，那么发送数据时就会自动在数据末尾发送所选择的校验位数据。



图 3-9 自动发送校验位设置

如图所示的例子。校验位选择了 CRC-16/MODBUS，然后发送 16 进制数据指令：011000000002040000，而实际发出去的指令末尾会自动增加 2 个字节的 CRC16 校验位数据，免去了用户自行计算添加校验位的麻烦。

3.4.5 发送文件数据

调试助手发送数据时，数据输入方式有两种，常规方式就是通过发送输入框输入数据；而在数据量比较大的情况下，可以将待发送的数据保存为文件，然后通过文件数据源发送数据。具体操作方法：在调试助手左侧控制面板的发送设置中，点击【打开文件数据源】；接着在弹出的文件选择对话框中选择待发送的目标文件，即载入文件数据源；最后，点击【发送】按钮，便开始从文件数据源发送数据。

3.4.6 循环发送设置

在发送设置面板中，设置好循环发送的时间周期，单位毫秒，并勾选【循环发送】选项，然后点击【发送】按钮，调试助手就会按照所设置的时间周期，循环发送输入的数据源。一旦启动循环发送过程，【发送】按钮便自动切换成【停止】按钮，要终止循环发送操作只要点击【停止】按钮即可。

3.4.7 回车键发送设置

调试助手的数据发送是通过鼠标点击【发送】按钮来触发的，也可以通过键盘按键的方式进行发送。默认情况下，单独按 Enter 键或者组合键 Ctrl+Enter 都可以执行数据发送操作，如果要在发送输入框中输入回车换行符，直接按回车键只能触发数据发送而无法输入回车换行，必须通过组合键 Shift+Enter 来插入回车换行符。

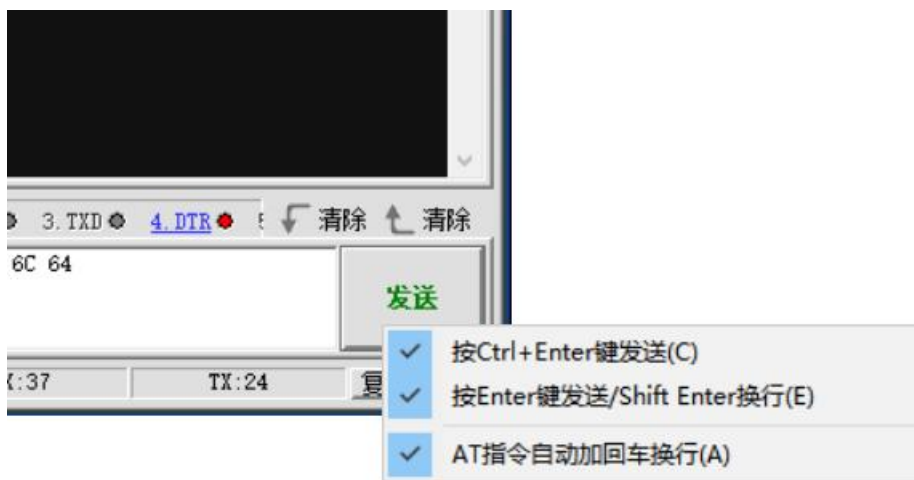


图 3-10 发送快捷键设置

如果要改变回车键的默认设置，可以右键点击【发送】按钮，在弹出的右键菜单中进行勾选设置，如上图 3-10 所示，具体选项说明如下。

- 1) [按 Ctrl+Enter 键发送]：该选项勾选后，组合键 Ctrl+Enter 执行发送操作；否则该组合键不执行任何操作；
- 2) [按 Enter 键发送/Shift Enter 换行]：该选项勾选后，单独按 Enter 键（不要组合 Shift 或 Ctrl）时执行发送操作，而组合键 Shift+Enter 用于在发送输入框中输入回车换行；如果取消勾选该选项，则单独按回车键时就会输入回车换行符而不会触发发送操作，并且组合键 Shift+Enter 无效。

3.5 发送框默认内容设置



图 3-11 发送框默认内容设置

调试助手软件启动后，发送输入框中的默认显示内容可以进行定制，可以是固定的数据内容（或为空），也可以记住上一次关闭时最后一条发送的数据。具体设置方法：在调试助手的菜单中找到【发送框中默认内容】选项，点击弹出设置窗口，如图 3-11 所示。可以选择【记住上一次成功发送的内容】或者【固定内容】。对于固定内容，可以选择 ASCII 码文本或者 HEX 码十六进制数据。修改设置后，点击确定按钮，下次启动调试助手软件时生效。

3.6 其它参数及控制选项

3.6.1 新建会话

在通信调试过程中，根据实际应用场景，可能随时需要运行多个调试助手软件。为了避免用户在电脑中查找/执行软件的麻烦，调试助手软件提供了【新建会话】功能，该控制选项位于系统菜单的第一个菜单项，用于启动另外一个调试助手的软件实例，也就是再运行一个调试助手软件。每点击一次【新建会话】选项，就会启动一个调试助手窗口进程，只要 PC 的内存足够大，可以同时启动任意多个调试助手窗口。

3.6.2 自动记住窗口大小

启用【自动记住窗口大小】菜单选项后，调试助手软件在每次关闭时都会自动保存记住当前的窗口大小，再次启动软件时，会自动恢复到上次关闭时软件窗口的大小。

3.6.3 窗口固定在最上层

【窗口固定在最上层】菜单选项，其实就是窗口置顶功能，可避免串口调试助手被其它软件窗口遮挡。该菜单选项还有一个快捷方式入口在调试助手主窗口标题栏右侧，是一个图钉样式的图标按钮，点击直接切换窗口置顶模式。

3.6.4 始终显示任务栏图标

调试助手软件启动后会在操作系统的任务栏条以及任务栏的托盘区显示软件图标。当调试助手最小化时，任务栏条上的图标在默认选项下会自动隐藏，只在任务栏的托盘区显示图标。如果用户的操作系统托盘区设置了折叠模式，那么调试助手的图标就有可能被隐藏起来，不容易直观地找到。这种情况下，就希望调试助手最小化时，不隐藏任务栏条上的软件图标。【始终显示任务栏图标】菜单项即用于实现该功能，该选项勾选后，操作系统的任务栏条上会始终显示调试助手的软件图标，方便用户定位工作窗口。

3.6.5 关闭按钮默认操作

点击调试助手主界面的关闭按钮，是执行软件关闭还是窗口最小化，可以通过该选项来设置，如下图所示。



图 3-12 关闭按钮默认操作设置

第四章 调试助手基本操作

本章介绍串口调试助手基本的操作使用方法，包括串口的设置、连接、串口数据的接收以及发送等。

4.1 串口参数设置

串口通信参数是指波特率、数据位、奇偶校验位和停止位这些参数。在串口设备之间进行连接通信时，通信双方必须约定使用完全相同的通信参数，才能建立有效的通信。也就是说，对于两个进行通信的串行端口，这些参数必须匹配，只有通信双方都在同一个频道上才能保证双方实现有效的数据沟通。

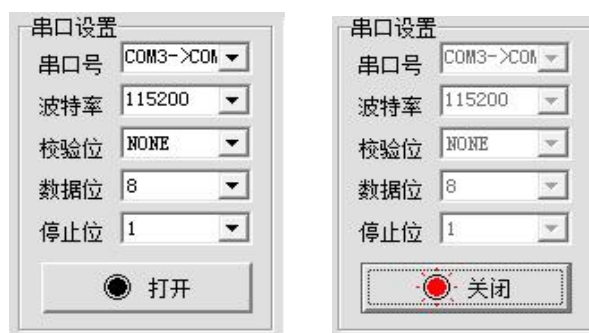


图 4-1 串口参数设置界面

(1) 波特率。这是一个衡量符号传输速率的参数。它表示每秒钟传送的符号的个数。例如 300 波特表示每秒钟发送 300 个符号。当我们提到时钟周期时，我们就是指波特率，例如如果协议需要 4800 波特率，那么时钟是 4800Hz。这意味着串口通信在数据线上的采样率为 4800Hz。通常电话线的波特率为 14400，28800 和 36600。波特率可以远远大于这些值，但是波特率和距离成反比。高波特率常常用于放置的很近的仪器间的通信，典型的例子就是 GPIB 设备的通信。

(2) 数据位。这是衡量通信中实际数据位的参数。当计算机发送一个信息包，实际的数据不会是 8 位的，标准的值是 5、7 和 8 位。如何设置取决于你想传送的信息。比如，标准的 ASCII 码是 0~127（7 位）。扩展的 ASCII 码是 0~255（8 位）。如果数据使用简单的文本（标准 ASCII 码），那么每个数据包使用 7 位数据。每个包是指一个字节，包括开始/停止位，数据位和奇偶校验位。由于实际数据位取决于通信协议的选取，术语“包”指任何通信的情况。

(3) 停止位。用于表示单个包的最后一位。典型的值为 1，1.5 和 2 位。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率同时也越慢。

(4) 奇偶校验位。在串口通信中一种简单的检错方式。有四种检错方式：偶、奇、高和低。当然没有校验位也是可以的。对于偶和奇校验的情况，串口会设置校验位（数据位后面的一位），用一个值确保传输的数据有偶个或者奇个逻辑高位。例如，如果数据是 011，那么对于偶校验，校验位为 0，保证逻辑高的位数是偶数个。如果是奇校验，校验位为 1，这样就有 3 个逻辑高位。高位和低位不是真正的检查数据，简单置位逻辑高或者逻辑低校验。

这样使得接收设备能够知道一个位的状态,有机会判断是否有噪声干扰了通信或者是否传输和接收数据是否不同步。

4.2 打开串口连接

建立串口通信前,先要确认目标串行通信设备是否已经电气连接到本地计算机串口(COM口或USB串口)端,并且需要知道设备所连接的计算机串口号。如果连接的是COM口,由于计算机的COM口都是有固定编号的,设备连接到几号COM口上,串口号就是几;如果目标通信设备连接的是USB转串口适配器(Dongle),那么串口号是不固定的,但我们可以通过插拔USB串口适配器,然后观察调试助手的串口号下拉列表中枚举的可用串口号的增减变化来确认目标设备连接的串口号。

在获悉目标通信设备所连接到本地计算机的串口号之后,就可以通过串口调试助手建立跟目标通信设备之间的通信连接了。操作方法很简单,将串口调试助手的串口设置参数中,选择串口号为目标设备一致的串口号,并将其他通信参数(波特率、数据位、奇偶校验位和停止位)设置成设备端一致,最后点击【打开】按钮。如果操作成功,按钮上指示灯由黑色转红色,表示串口成功打开,串口通信建立完成。

4.3 串口数据接收

串口调试助手在打开串口后,不需要其它额外操作(按默认设置),就可以接收来自串口发送过来的数据,接收到的数据会按照接收设置中设定的模式显示在接收窗口。如下图所示:

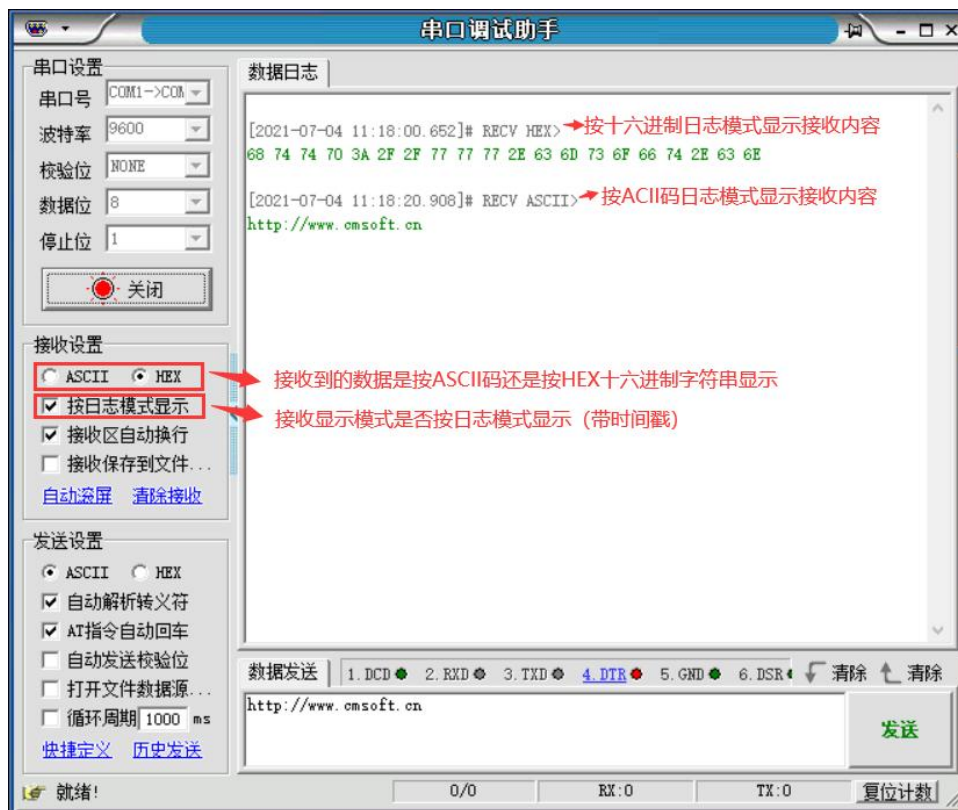


图 4-3 串口数据接收

为了方便对接收数据的观察分析，可以在接收设置中，对接收模式进行一些修改调整。最常用的是设置数据类型（ASCII/HEX），即接收到的数据是按 ASCII 码字符串显示还是按 16 进制字符串显示。如果接收到的数据是文本内容，那么就选择 ASCII 方式；如果接收到的数据是二进制数据或者包含非打印字符，那么就选择 HEX 模式，也就是将接收到的数据转换成十六进制字符串后显示。

另外，接收设置中有一个常用选项叫做“按日志模式显示”。如果不勾选这个选项，那么接收窗口只显示接收到的数据，不会显示送的数据，并且显示接收数据时也不带时间戳；如果勾选了这个选项，那么，不管是接收或者发送的数据都会显示在窗口中，并且每条数据记录都带时间戳。

4.4 串口数据发送

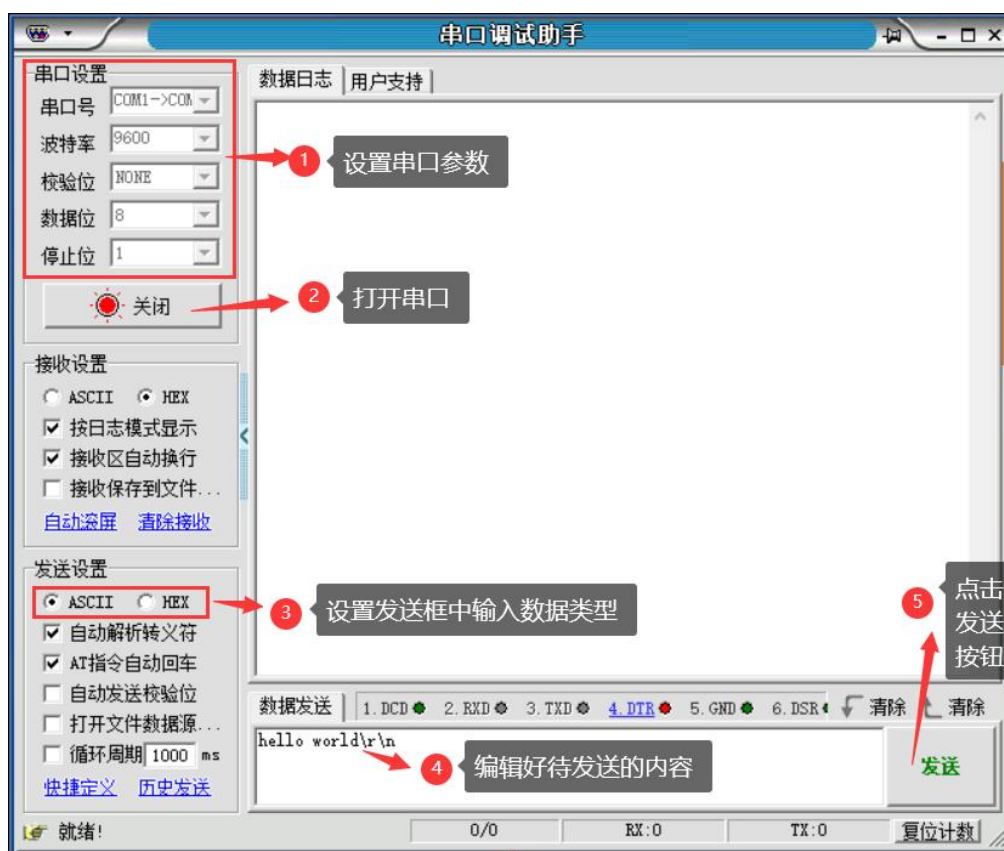


图 4-2 串口数据发送流程

串口调试助手往目标串口发送数据，共分为 5 个步骤：

- ① 串口参数设置
- ② 打开串口连接
- ③ 设置发送的数据类型（ASCII 码字符串/HEX 十六进制字符串）
- ④ 编辑好待发送的内容
- ⑤ 点击发送按钮

4.5 发送转义字符

以 ASCII 码字符串方式发送数据时,允许用户在字符串中使用转义字符的方式插入非打印字符。最简单的例子,发送一条带回车换行结尾的 AT 指令,只要在发送框输入 AT\r\n,然后直接点击发送按钮即可。

目前,调试助手支持如下这些 C 语言标准转义字符:

转义字符	含义
\r	回车符,等价十六进制 \x0D
\n	换行符,等价十六进制 \x0A
\t	水平制表符(HT),等价十六进制 \x09
\v	垂直制表符(VT),等价十六进制 \x0B
\a	响铃(BEL),等价十六进制 \x07
\b	退格符(BS),等价十六进制 \x08
\f	换页符(FF),等价十六进制 \x0C
\xhh	2 位十六进制转义字符
\ddd	3 位八进制转义字符

作为扩展,调试助手支持转义十六进制数组,例如: \xA0\x12\xF1\xAB\xCD\x51\xF3 等价于 \x[A0 12 F1 AB CD 51 F3]。包含在中括号[]中的多个 16 进制字节之间可以使用若干空格符分割或没有空格。

注意:使用转义符时,必须勾选主界面左侧发送设置中的【自动解析转义符】选项,否则调试助手不会对转义符进行任何解析处理。

4.6 发送指令脚本

通过转义符扩展,调试助手在 V5.0.2 版本之后开始支持发送指令脚本,允许用户在发送的指令数据中,加入各种业务处理逻辑,嵌入包含函数以及计算表达式的脚本代码,动态计算生成最终用于发送的数据内容。出于调试目的,用户还可以在发送的指令中调用 printf 函数进行调试打印,调试输出结果会显示在日志窗口。

下面这个例子,通过调试助手发送一条 Modbus 指令。在调试助手的发送窗口输入以下内容: \x[01 04 00 00 00 04]\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]

这条指令表示发送一组长度为 6 字节的十六进制数据 01 04 00 00 00 04,后面跟 2 字节的 CRC16 校验码,这个校验码通过以下代码动态计算获得:

```
\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]
```

其中,\[]称为模式符,用作嵌入脚本代码的容器。上述表达式通过冒号分割成两部分,冒号前的 2 表示最终计算值只取 2 字节,冒号后的表达式用于计算校验码。表达式中的 calculate 是系统内置函数,用于计算校验算法,calculate 函数的第 1 个参数表示从当前发送数据的第几个字节开始计算校验码;第 2 个参数表示校验数据长度,此长度可以为负数,比如为-1 时,表示数据长度截止到当前 calculate 函数调用位置的前一个字节数据,-2 则表示数据长度从当前位置往前推 2 个字节,以此类推;calculate 函数的第 3 个参数表示使用的算法,ALGO_CRC16_MODBUS 是系统内置常数,表示 MODBUS_CRC16 校验算法。函数 reverse 用于将目标数据的字节顺序进行逆转(高低字节交换重排)。这里调用 reverse 的目的是因为 calculate 函数计算出的 16 位 CRC 校验码是网络字节顺序(BigEndian),但是 ModbusRTU

协议中的 CRC 校验码要求使用 LittleEndian 字序，所以这里要进行字节顺序反转处理。



图 4-3 发送包含函数表达式的指令脚本

发送脚本代码时，必须勾选发送设置中的【ASCII】模式以及【转义字符指令解析】这两个选项。这是因为，只有选择【ASCII】才允许输入脚本代码，否则若选择 HEX 模式，那么就只能输入十六进制数字；并且必须勾选【转义字符指令解析】选项后，调试助手才会对通过反斜杠导入的脚本表达式进行解析。

指令中嵌入脚本代码必须使用模式符\[]。具体的嵌入方式有两种：运算表达式和 BLOCK 代码块。

4.6.1 运算表达式

这里的运算表达式，特指具有返回值的基于类 C 语言语法规则的计算表达式。其一般形式为：`\[n:expression#remark]`

这是一种三段式表示法：第 1 段是输出数据长度，第 2 段是计算表达式，第 3 段是注解（注释）文字。其中第 1 段和第 3 段都可以省略，最简形式为 `\[expression]`。

如果设置的输出长度(n)大于实际计算表达式(expression)的最终计算值的长度，则用 0 补足后输出，反之若设置的输出长度小于最终计算结果的长度，则截掉高位多余字节后输出。如果省略第 1 段的长度值，则按表达式计算结果的固有长度输出。注解字段为可选字段，以#号开头，表示注释性文字，也可以作为注解名被引用。

例如,上一小节示例中发送的内容中包含的单行表达式字段如下:

```
\[2:reverse(0,-1,ALGO_CRC16_MODBUS)]
```

如果进一步省略输出长度,可简化为

```
\[reverse(0,-1,ALGO_CRC16_MODBUS)]
```

虽然调用了多个函数,多个参数,但是只有仅仅 1 条语句(1 个表达式),并有 1 个返回值。

4.6.2 BLOCK 代码段

BLOCK 块级代码字段的形式为 `\[{ script }]`

块级代码可以包含任意多条语句,最终的计算结果通过 echo 或 echob 函数产生,或者通过 return 语句返回。例如,上例中的单表达式字段,可以改写成以下的块级代码段:

```
\{  
short crc16=calculate(0,-1,ALGO_CRC16_MODBUS);  
crc16=reverse(crc16);  
return crc16; //或者 echo(crc16);  
}
```


第五章 调试助手进阶选项

调试助手的工具面板窗口提供通信调试的进阶操作选项。包括快捷指令、批量发送、自动应答、历史发送、校验计算器等。

5.1 快捷指令

调试助手软件在跟目标设备或串口应用服务进行通信调试时,通常需要根据实际应用协议,发送一些交互指令或数据。当指令比较多时,现场输入这些指令数据肯定会麻烦且低效。针对这个问题,本调试助手软件提供了指令预定义功能,也就是工具面板中的【快捷指令】(老版本中称之为“快捷定义”)。该功能支持按十六进制或 ASCII 码字符串的方式进行指令预定义,可以定义指令的名称以及对应的数据内容,并可定义发送热键(快捷键)。预定义的指令可以直接导出保存为文件,也可以随时从外部文件导入。默认情况下,预定义指令会自动以配置文件的形式保存在当前应用程序目录下,在软件启动时会自动载入。如下图所示。



图 5-1 快捷指令/预定义指令列表

如图 5-1 所示,每条预定义指令都包含名称、数据、快捷键三个部分。指令名称相当于指令备注,对指令数据起到描述作用;每条指令都可以单独设置快捷键,通过键盘按下快捷键可以直接发送对应的指令数据。每条指令,不管是否定义快捷键,都可以直接通过点击快捷键栏中的按钮,或者通过回车键发送选中的指令数据。

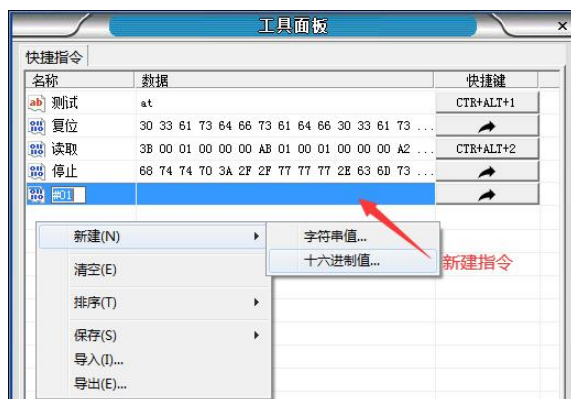


图 5-2 新建预定义指令

通过右键菜单可以对预定义指令进行增、删、改、排序、导入、导出等操作。以新建预定义指令为例，右键点击【快捷指令】窗口的空白处，弹出右键菜单，选择【新建】菜单项，就可以新建预定义指令。新建指令时，根据实际应用场景，选择字符串值还是十六进制值，然后在快捷指令窗口完成指令名称及数据内容的输入。

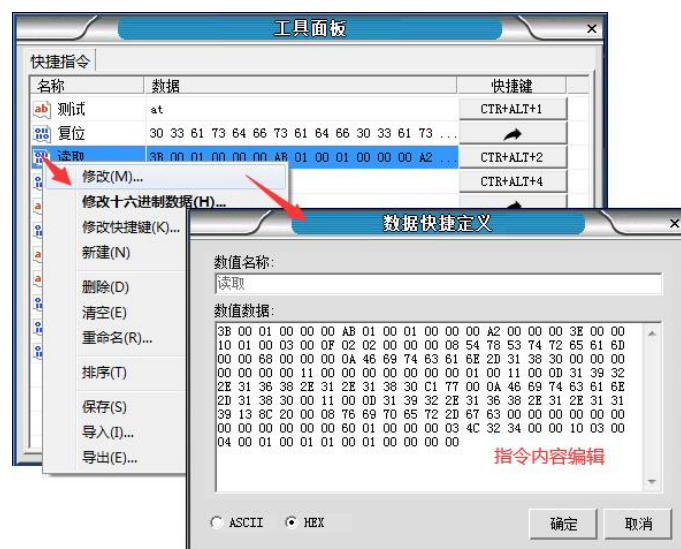


图 5-3 编辑预定义指令

如果要编辑已经输入的预定义指令，可以直接右键点击目标指令，然后在弹出的右键菜单中选择对目标指令进行编辑，包括修改数据内容、数据类型（ASCII/HEX）、设置快捷键、重命名、删除等操作。例如上图 5-3，是对一条指令进行数据内容的编辑，编辑完点击【确定】按钮关闭编辑窗口，编辑修改过的内容会在软件退出时自动保存。

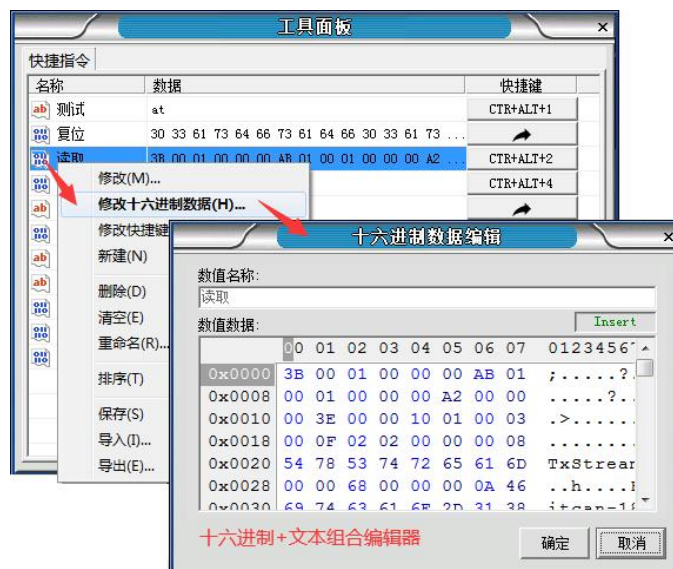


图 5-4 使用十六进制专用编辑器修改指令

图5-4所示，选中目标指令后，可以通过右键菜单选择【修改十六进制数据】，使用十六进制专用编辑器，以十六进制与ASCII码相互对照的方式对数据指令进行编辑。

5.2 批量发送

批量发送功能用于控制多条预定义数据指令按照一定的顺序和不同的延迟间隔进行发送。【批量发送】功能位于工具面板窗口，类似【快捷指令】功能，需要先进行预定义指令的编辑输入。每条指令除了可以定义指令数据和文字备注，还定义有延迟时间，即上一条指令发送完后，延迟多长时间发送当前这条指令。



图 5-5 批量发送~指令列表

如图 5-5 所示，批量发送窗口中，每条预定义指令包含编号、延迟时间、数据、及备注四个部分。指令编号顺序决定了批量发送的指令顺序，并且只有前面复选框被勾选的指令才会被批量发送；延迟是指发送前延迟，也就是当前指令与前一条发送的指令之间的发送延迟时间，单位毫秒；指令备注实际上类似于【快捷指令】中的指令名称，用于对指令起注解作用，点击备注栏中的按钮可以直接发送对应的指令数据。

勾选批量发送窗口右下角的【开始发送】选项，调试助手就会依次发送指令列表中勾选的指令，在发送每条指令前都会执行设置的延迟等待时间。发送完所有的指令，自动停止发送，并自动取消【开始发送】复选框。如果勾选了【循环模式】选项，批量发送会周而复始的进行，直至手动取消勾选【循环模式】或者【开始发送】选项。

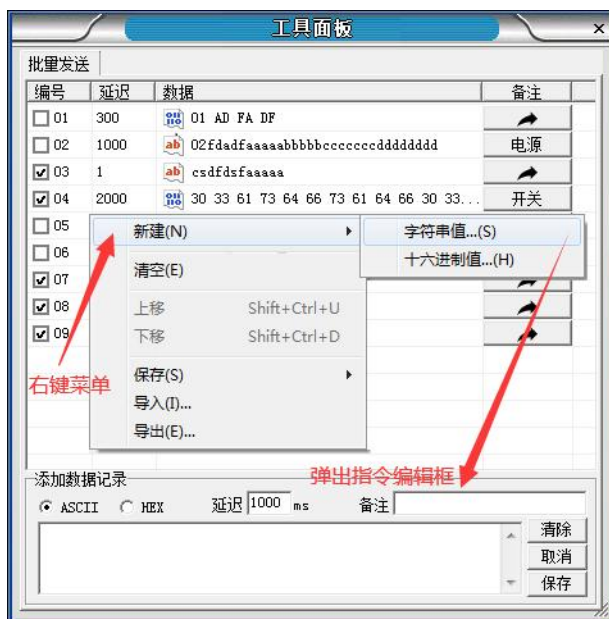


图 5-6 新建批量发送指令

通过右键菜单，可以对批量发送的指令进行增、删、改、排序、导入、导出等操作。以新建批量指令为例，右键点击【批量发送】窗口的空白处，弹出右键菜单，选择【新建】菜单项，再根据实际应用场景，选择字符串值还是十六进制值，最后在弹出的编辑窗口完成指令内容的输入，如图 5-6 所示。编辑完内容后，点击保存按钮，如果要再次编辑这条指令，可以直接在指令列表中双击目标指令，或者右键点击目标指令，然后在弹出的右键菜单中点击【修改】菜单项，即会弹出目标指令的编辑窗口。

5.3 自动应答

自动应答功能，用于实时对调试助手接收到的数据指令进行匹配识别，并自动按用户预定义规则发送相应的应答指令。用户只须事先设计好应答规则，然后调试助手内部集成的规则引擎会自动对接收到的数据进行指令规则匹配及应答数据的发送。如果需要同时支持多条指令的自动应答，只要相应地建立多条自动应答规则。

自动应答是在 V5.0.1 版本之后新增加的特色功能，自动应答的具体规则将在本文第七章中详细介绍。

5.4 历史发送

工具面板中的历史发送窗口，记录着用户发送的历史指令数据。用户可随时调出历史发送窗口进行查看，如下图5-7所示。



图 5-7 历史发送记录

每一条历史发送记录的组成都分数据内容及记录时间两部分，其中记录时间就是该数据的初次发送的时间，该时间以按钮的形式显示，点击该按钮可以直接重发这条指令。但重发的指令不会历史发送窗口增加新的发送记录，除非删除存在的历史记录。实际上，历史发送窗口，只保存主窗口中通过发送按钮发送的数据，而通过快捷指令、批量发送、历史发送这些方式发送的数据是不会保存到历史发送记录中去的。

5.5 校验计算器

本调试助手软件支持发送数据时自动添加校验码,也可以自行通过调试助手提供的校验计算器来进行数据的校验码计算,然后手动添加到发送数据中。

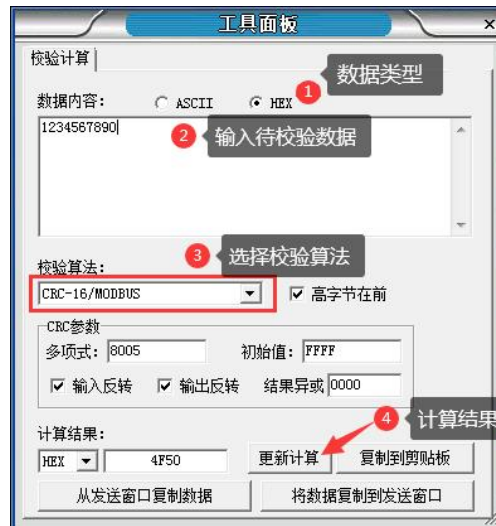


图 5-8 校验位计算器

校验位计算器从工具面板中打开,如图 5-8 所示。检验的数据类型可以选择 ASCII 码或者 HEX 码,但数据类型必须跟实际输入的数据内容一致;可供选择的主要校验算法如下:

- CHECKSUM-8: 8 位 (1 字节) 累加和校验码,校验码为校验数据所有字节之和。
- CHECKSUM-8N/LRC: 纵向冗余校验 (LRC, Longitudinal Redundancy Check), 校验值为校验数据所有字节之和的负数。
- XOR-8/BCC: 信息组校验 (BCC, Block Check Character), 校验值为校验数据所有字节的异或值。
- CRC-8: 8 位 CRC 校验, 多项式 $x^8 + x^2 + x + 1$ (0x07), 初始值 0x00, 输入数据不反转, 输出数据不反转, 输出结果异或值 0x00。
- CRC-16/MODBUS: 适用于 Modbus 的 16 位 CRC 校验码, 多项式 $x^{16} + x^{15} + x^2 + 1$ (0x8005), 初始值 0xFFFF, 输入数据反转, 输出数据反转, 输出结果异或值 0x0000。
- CRC-16/CCITT: 多项式 $x^{16} + x^{12} + x^5 + 1$ (0x1021), 初始值 0x0000, 输入数据反转, 输出数据反转, 结果与 0x0000 异或。
- CRC-16/CCITT-FALSE: 多项式 $x^{16} + x^{12} + x^5 + 1$ (0x1021), 初始值 0xFFFF, 输入数据不反转, 输出数据不反转, 结果与 0x0000 异或。
- CRC-16/XMODEM: 多项式 $x^{16} + x^{12} + x^5 + 1$ (0x1021), 初始值 0x0000, 输入数据不反转, 输出数据不反转, 结果与 0x0000 异或。
- CRC-16/X25: 多项式 $x^{16} + x^{12} + x^5 + 1$ (0x1021), 初始值 0xFFFF, 输入数据反转, 输出数据反转, 结果与 0xFFFF 异或。
- CRC-16/IBM: 多项式 $x^{16} + x^{15} + x^2 + 1$ (0x8005), 初始值 0x0000, 输入数据反转, 输出数据反转, 结果与 0x0000 异或。
- CRC-16/MAXIM: 多项式 $x^{16} + x^{15} + x^2 + 1$ (0x8005), 初始值 0x0000, 输入数据反转,

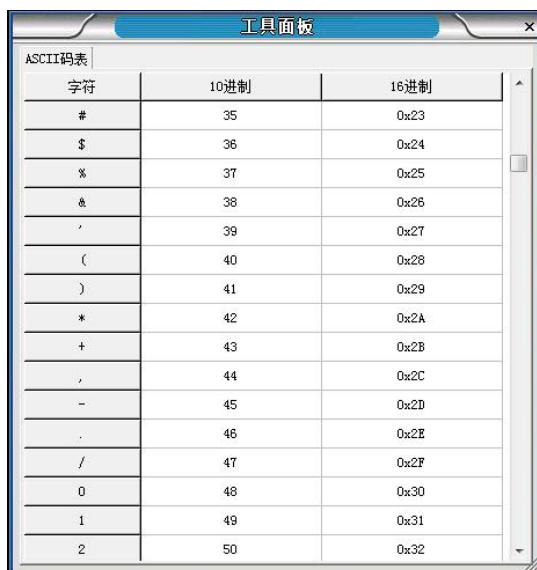
输出数据反转，结果与 0xFFFF 异或。

- CRC-16/USB: 多项式 $x^{16}+x^{15}+x^2+1$ (0x8005)，初始值 0xFFFF，输入数据反转，输出数据反转，结果与 0xFFFF 异或。
- CRC-32: 多项式 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (0x04C11DB7)，初始值 0xFFFFFFFF，输入数据反转，输出数据反转，结果与 0xFFFFFFFF 异或。
- CRC-32/BZIP2: 多项式 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (0x04C11DB7)，初始值 0xFFFFFFFF，输入数据不反转，输出数据不反转，结果与 0xFFFFFFFF 异或。
- CRC-32/MPEG-2: 多项式 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (0x04C11DB7)，初始值 0xFFFFFFFF，输入数据不反转，输出数据不反转，结果与 0x00000000 异或。
- MD5: 16 字节 MD5 校验码。
- FIXED HEX BYTES: 固定数据校验码，校验码为任意多字节常数；

CRC 校验算法，支持 CRC8、CRC16、CRC32 三种宽度的校验算法，并可自定义修改校验参数，如多项式、初始值、输入数据翻转、输出数据翻转、输出结果异或值。通过修改 CRC 参数，可以实现任意的 CRC 校验算法。

5.6 ASCII码对照表

ASCII (American Standard Code for Information Interchange) 是基于拉丁字母的一套电脑编码系统, 它主要用于显示现代英语和其他西欧语言。到目前为止共定义了 128 个字符, 其中包含 33 个控制字符 (具有某些特殊功能但是无法显示的字符, 也就是非打印字符) 和 95 个可显示字符 (可打印字符)。



字符	10进制	16进制
#	35	0x23
\$	36	0x24
%	37	0x25
&	38	0x26
'	39	0x27
(40	0x28
)	41	0x29
*	42	0x2A
+	43	0x2B
,	44	0x2C
-	45	0x2D
.	46	0x2E
/	47	0x2F
0	48	0x30
1	49	0x31
2	50	0x32

图 5-9 ASCII 对照表

调试助手提供 ASCII 对照表，如图 5-9 所示。可帮助用户在通信调试时对收发数据中字符数据解析处理。比如，接收到 16 进制数据后，若需要手动转换成 ASCII 码字符串时，可

以借助 ASCII 码表逐个根据十六进制值查找目标字符；再比如，发送包含非打印字符的 ASCII 字符串时，可以通过转义符的方式，即反斜杠加小写字母 x 再加两位 16 进制数据的表示方法，而非打印字符对应的 16 进制值可以通过 ASCII 码表来查找。

5.7 命令行启动参数

调试助手可以通过命令行参数，按给定的参数选项启动，还可以直接通过命令行发送串口数据。UART 串口通信模式时，命令行参数格式设置方法如下：

```
uartassist.exe [-p 串口号] [-b 波特率] [-c 校验位] [-w 数据位] [-s 停止位] [-x] [-d 数据内容]
```

参数说明：

- 串口号：如 COM1、COM2、COM3...
- 波特率：填写实际使用的波特率，如 9600、19200、115200...
- 校验位：取 NONE、ODD、EVEN、MARK、SPACE；
- 数据位：取 5、6、7、8；
- 停止位：取 1、1.5、2；
- 数据内容：-d 参数后跟待发送的数据内容，一定要包含在双引号内，若加参数-x 则表示按 HEX 格式发送，否则按 ASCII 码格式发送。

例 1: `uartassist.exe -p COM6 -b 115200`

命令行启动调试助手软件，并打开串口 COM6，波特率设置为 115200，其他未设置的参数取默认值。注意，命令行方式启动调试助手软件时，如果指定了串口号，就会自动尝试打开目标串口。

例 2: `uartassist.exe -p COM6 -b 115200 -c NONE -w 8 -s 1 -d "hello" -q`

命令行启动调试助手发送字符串数据，执行完毕后自动关闭调试助手软件。参数-q 表示执行完退出调试助手进程。

说明：通过调试助手命令行参数发送数据时，由于只能携带文本字符串参数，不能直接发送二进制数据，但是可以通过转义字符方式实现二进制数据的发送。比如发送一条带回车换行符结尾的 AT 指令，对应数据 `-d "AT\r\n"`

等价于 `-d "AT\x0D\x0A"`

等价于 `-d "\x41\x54\x0D\x0A"`

等价于 `-d "\x[41 54 0D 0A]"`

等价于 `-d "\x[41540D0A]"`

等价于 `-x -d "41540D0A"`

等价于 `-x -d "41 54 0D 0A"`

注意，带-x 参数时，在-d 参数后面直接跟十六进制数据，不用转义字符。

实际执行命令：`uartassist.exe -p COM6 -b 115200 -q -d "AT\r\n"`

等价于：`uartassist.exe -p COM6 -b 115200 -q -x -d "41540D0A"`

第六章 脚本代码语法规则

调试助手的指令数据或指令模板可以嵌入类 C 语言语法的脚本代码，方便用户灵活地编写指令或模板，实现强大的高级指令发送或者自动应答功能。

6.1 运算符

自动应答规则引擎，支持各种逻辑运算及位操作符。一共有 34 种运算符，10 种运算类型：算术运算符（+、-、*、/、%）、关系运算符（>、>=、==、!=、<、<=）、位运算符（>>、<<、==、!=、<、<=）、逻辑运算符（!、||、&&）、条件运算符（?:）、指针运算符（&、*）、赋值运算符（=）、逗号运算符（,）、强制类型转换运算符（（类型名））、其他（下标[]、分量、函数）；若按参与运算的对象个数，运算符可分为单目运算符（如!）、双目运算符（如+、-）和三目运算符（如?:）。指令模板中实际常用的运算符，如下表所示：

优先级	运算符	名称或含义	使用形式	说明
1	()	圆括号	(表达式)、函数名(形参表)	
2	(类型)	强制类型转换	(数据类型)表达式	
	-	负号运算符	-表达式	单目运算符
	!	逻辑非运算符	!表达式	单目运算符
	~	按位取反运算符		单目运算符
3	/	除	表达式 / 表达式	双目运算符
	*	乘	表达式*表达式	双目运算符
	%	余数（取模）	整型表达式%整型表达式	双目运算符
4	+	加	表达式+表达式	双目运算符
	-	减	表达式-表达式	双目运算符
5	<<	左移	变量<<表达式	双目运算符
	>>	右移	变量>>表达式	双目运算符
6	>	大于	表达式>表达式	双目运算符
	>=	大于等于	表达式>=表达式	双目运算符
	<	小于	表达式<表达式	双目运算符
	<=	小于等于	表达式<=表达式	双目运算符
7	==	等于	表达式==表达式	双目运算符
	!=	不等于	表达式!=表达式	双目运算符
8	&	按位与	表达式&表达式	双目运算符
9	^	按位异或	表达式^表达式	双目运算符
10		按位或	表达式 表达式	双目运算符
11	&&	逻辑与	表达式&&表达式	双目运算符
12		逻辑或	表达式 表达式	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2:表达式 3	三目运算符

6.2 运算表达式

所谓运算表达式，就是具有返回值的基于类 C 语言语法规则的计算表达式。

例如：

➤ \[2: 2*(getuchar(0)+getuchar(1))]

➤ `\[(getuchar(#len)+1) : 2*(getuchar(0)+getuchar(1))]`

在指令自动匹配应答过程中，运算表达式会实时动态地被最终计算值所替换。因此，运算表达式在形式上必须是一条具有返回值的脚本语句构成，而不允许分多条语句实现。如果一条语句无法实现，只能通过下一节所介绍的包含多条脚本语句的 BLOCK 代码段实现。BLOCK 代码块的返回值通过 `return` 语句或者 `echo/echob` 函数实现传递。

6.3 BLOCK 代码块

自动应答规则的指令模板，可以要使用包含多条语句的 BLOCK 语法。一个 BLOCK 的多条语句通过大括号对 `{}` 包括起来，多条语句之间用分号隔开，最终通过 `return` 语句，或者 `echo` 函数返回 BLOCK 表达式的最终值。不同的是，`return` 返回值后就会终止当前 BLOCK 后面的语句，而 `echo` 返回值之后会继续执行 BLOCK 后面的语句。如果一个 BLOCK 中执行了多次 `echo`，则每次返回的数据会追加到之前返回数据的后面。如果没有执行到 `echo` 或 `return`，则表示无返回数据；如果既有 `echo` 值，又有 `return` 值，则 `echo` 值会忽略而只取 `return` 值。

例如，下面这个 BLOCK 例子：

```
\[{
    int num=getuchar(#num);
    if(num>0){
        echo(0xF2F1);
        echo("\xF3\xF4%x", num);
    }
    else return 0;
}]
```

代码说明：若 `if(num>0)` 表达式成立，则调用两次 `echo`，两次数据叠加，最终返回 16 进制数据流 F1 F2 F3 F4 num；若 `if(num>0)` 表达式不成立，则返回数值 0。

作为扩展，应答规则中脚本代码支持 0x 开头的 16 进制立即数，如上面代码中的 0xF2F1 就是一个 16 进制立即数。

6.4 变量数据类型

自动应答规则引擎的内置脚本代码只能使用下表所示的基本数据类型，不支持用户自定义变量结构体。目前所支持的基本数据类型如下：

类型名	可用别名	类型说明
<code>char</code>		有符号字符
<code>unsigned char</code>	<code>byte</code> 或 <code>uchar</code>	无符号字符
<code>short</code>		有符号短整形
<code>unsigned short</code>	<code>ushort</code>	无符号短整形
<code>int</code>		有符号整形（32 位）
<code>unsigned int</code>	<code>uint</code>	无符号整形（32 位）
<code>float</code>		32 位浮点数
<code>bool</code>		布尔类型
<code>string</code>		字符串类型

使用限制：仅支持一维数组和一维指针，暂不支持多维数组和多维指针；暂不支持 64 位数据类型。

6.5 变量定义及作用域

自动应答规则的脚本语言支持使用变量。根据变量的强弱类型可划分为，强类型变量和弱类型变量；根据变量作用域，则可分为局部变量及全局变量。

6.5.1 强类型变量与弱类型变量

(1) 强类型变量。强类型变量类似于标准 C 语言的变量定义方式，必须先定义后使用。强类型变量的数据类型在变量定义时就被指定，不允许动态修改变量类型。强类型变量定义时，需要指定变量的数据类型以及变量名，并且允许在变量定义时初始化赋值。如：

```
\[{ int x,y; //定义两个强类型的整形变量
    int z=100; //定义一个初值为 100 的强类型整形变量
    char *str1="abc"; //定义一个 null-terminated 字符串
    string str2="abc\x00\x01\x02"; //定义一个标准字符串，允许包含 0。
}]
```

(2) 弱类型变量。弱类型变量无需声明或定义，也不用指定变量的数据类型，可以直接使用。给弱类型变量赋值时，如果变量名不存在则会自动创建该变量。弱类型变量的数据类型总是等于最后一次赋值的数据类型。也就是说，弱类型变量的数据类型是可以被动态修改的。弱类型变量必须通过保留字 `global` 按数组索引的方式来使用，数组的下标为字符串形式的变量名。例如：

```
\[ {
    global["x"]=100; //给弱类型全局变量 x 赋值整形数 100。
    global["x"]="abcdefg"; //给弱类型全局变量 x 赋值字符串。
}]
```

6.5.2 局部变量与全局变量

自动应答规则中的变量，如果是强类型方式定义的，只能作为局部变量，仅在当前指令模板的当前 `\[]` 段内有效。多个 `\[]` 内的同名局部变量互不影响。

如果需要使用全局变量，就必须使用 `global["name"]` 形式的弱类型变量。弱类型变量的作用域覆盖所有指令模板，在调试助手整个运行期间都常驻保留在内存中。

弱类型全局变量一旦初始化赋值后，就可以省略 `global` 关键字，而直接通过变量名访问，就跟操作强类型变量一样。仅当需要修改变量数据类型时，才必须通过 `global` 关键字来给变量赋值。

6.6 变量强制类型转换

跟标准的 C/C++ 语法规则一样，当操作数的类型不同，经常需要将操作数转化为所需要的类型，这个过程即为强制类型转换。

6.6.1 强制类型转换的形式

变量强制类型转换具有两种形式：显式强制类型转换和隐式强制类型转换。下面就两种形式分别进行简单的描述。

(1) 显式强制类型转换

显式强制类型转换很简单，格式为：TYPE b = (TYPE) a;

其中，TYPE 为类型描述符，如 int, float 等。经强制类型转换运算符运算后，返回一个具有 TYPE 类型的数值，这种强制类型转换操作并不改变操作数本身，运算后操作数本身未改变，例如：

```
int n=0xab65;
char a=(char) n;
```

上述强制类型转换的结果是将整型值 0xab65 的高端一个字节删掉，将低端一个字节的內容作为 char 型数值赋值给变量 a，而经过类型转换后 n 的值并未改变。

(2) 隐式强制类型转换

隐式类型转换发生在赋值表达式和有返回值的函数调用表达式中。在赋值表达式中，如果赋值符左右两侧的操作数类型不同，则将赋值符右边操作数强制转换为赋值符左侧的类型数值后，赋值给赋值符左侧的变量。在函数调用时，如果 return 后面表达式的类型与函数返回值类型不同，则在返回值时将 return 后面表达式的数值强制转换为函数返回值类型后，再将值返回，如：

```
int n;
double d=3.88;
n=d; //执行本句后，n 的值为 3，而 d 的值仍是 3.88。
```

6.6.2 强制类型转换在自动应答规则中的典型用途

在指令应答模板中的模式应答数据段，如果不显式指定数据长度，则默认长度为其数据类型的固有长度。

比如，有一个整形数据段，默认长度为 4 字节。如要要求只取 2 个字节。可以显示指定数据长度为 2，或者强制类型转换为 short 类型。例如，以下两个指令应答模板等价：

```
\[2:getshort(0)*100]           //直接指定数据长度 2
\[ (short) (getU16(0)*100)]     //强制类型转成 2 字节的 short 数据类型
```

注：数值计算表达式的返回值，如果不做强制类型转化，则默认数据类型是 32 位的 int 或者 float 类型，长度 4 字节。

6.7 语法大小写规则

调试助手内置脚本代码是不区分大小写的。但是为避免混乱，推荐大家始终坚持“大小写敏感”的代码书写规范，尽量保证函数名、变量名、常数名等大小写前后书写一致。

6.8 字段注解的定义及引用

不管是脚本指令还是指令模板（包括指令匹配模板及指令应答模板）都可以包含若干个

模式段，每个模式段的一般形式都可以归结为：

`\[exp_len:exp_value#comment]`

其中，exp_len 为字段数据长度、exp_value 为字段数据内容、#comment 为字段注解。

注解的形式为#号开头加注解文字。注解字段的作用有两个：

(1) 起注释作用，对目标字段作解释说明。

(2) 给字段命名，经过命名的字段数据可以被同一个指令模板的其他模式段通过注解名引用。引用注解只能通过以下这几个用于读取指令数据段的系统函数：gets、getchar、getuchar、getshort、getushort、getint、getuint。除此之外，其他函数都不支持调用注解作为参数。

下面举个简单例子作说明：

➤ 指令匹配模板：REQ\[2 #command]

➤ 指令应答模板：ACK\[2:gets(#command, 2)]

本例指令匹配模板中，\[2 #command] 定长模糊匹配 2 个字节的 command 数据；而在应答模板中需要复制这两个字节内容作为应答数据帧的一部分。在应答模板对应的模式应答段中，通过 gets(#comment, len) 函数可以读取注解名对应的数据块，省略 gets 的第二个参数表示读取整个目标字段。本例中，应答模板需要复制请求指令中整个 command 段的数据，因此可以省略 gets 函数的第 2 个表示长度的参数。这样，此应答模板可以简化为：

➤ 指令应答模板→简化：ACK\[2:gets(#command)]

如果模式应答段需要的数据长度跟 gets 返回的数据块长度一致，就可以省略模式应答段的长度参数，如下所示：

➤ 指令应答模板→再简化：ACK\[gets(#command)]

6.9 内建系统函数详解

自动应答规则的运算表达式，可以调用规则引擎内建的系统函数(函数名不区分大小写)，目前支持的函数如下。

(1) printf - 格式化输出到控制台

函数原型：void printf(const char format, 可选参数...);

功能描述：按指定格式向标准输出设备（日志/接收窗口）输出调试打印信息。

(2) sprintf - 格式化字符串

函数原型：int sprintf(char *buffer, char *format [, argument, ...]);

功能描述：把格式化的数据写入某个字符串缓冲区。如果成功，则返回写入的字符总数，不包括字符串追加在字符串末尾的空字符。如果失败，则返回一个负数。

(3) strtoint - 字符串转整形数

函数原型：int strtoint(const char *nptr);

函数别名：atoi

功能描述：strtoint 函数会扫描参数 nptr 字符串，跳过前面的空白字符（例如空格，

tab 缩进)等,扫描直至下一个非数字字符或到达输入的结尾。如果 `nptr` 不能转换成整数,那么将返回 0。

(4) `inttostr` - 整形数转字符串

函数原型: `string inttostr(int n);`

功能描述: 将整形数 `n` 转换成字符串类型返回。

(5) `strcpy` - 字符串拷贝

函数原型: `char *strcpy(char* dest, const char *src);`

功能描述: 把 `src` 指向的 null-terminated 字符串复制到 `dest` 指向的地址空间,并返回指向 `dest` 的指针。

(6) `strcat` - 字符串拼接

函数原型: `char *strcat(char *dest, const char *src1, const char *src2, ...);`

功能描述: 把若干个 NULL 结尾的源字符串 (`src1`、`src2`、...) 复制拼接到 `dest` 所指向的字符串后面(删除 `dest` 原来末尾的“\0”)。要保证 `dest` 指向的字符串空间足够长,以容纳被复制进来的源字符串。最后返回指向 `dest` 的指针。

(7) `strcmp` - 字符串比较(区分大小写)

函数原型: `int strcmp(const char *s1, const char *s2);`

功能描述: 两个字符串自左向右逐个字符相比(按 ASCII 值大小相比较),直到出现不同的字符(区分大小写),或遇'\0'为止。当 `s1<s2` 时,返回为负数;当 `s1=s2` 时,返回值=0;当 `s1>s2` 时,返回正数。

(8) `stricmp` - 字符串的比较(不区分大小写)

函数原型: `int stricmp(const char *s1, const char *s2);`

功能描述: 两个字符串自左向右逐个字符相比(按 ASCII 值大小相比较),直到出现不同的字符(不区分大小写),或遇'\0'为止。当 `s1<s2` 时,返回为负数;当 `s1=s2` 时,返回值=0;当 `s1>s2` 时,返回正数。

(9) `strncpy` - 限定长度的字符串拷贝

函数原型: `char *strncpy(char* dest, const char *src, int n);`

功能描述: 把 `src` 指向的 null-terminated 字符串复制到 `dest` 指向的地址空间,并返回指向 `dest` 的指针。如果源字符串实际长度大于参数 `n`,则最多复制 `n` 个字节。

(10) `strncmp` - 限定长度的字符串比较(区分大小写)

函数原型: `int strncmp(const char *s1, const char *s2, int n);`

功能描述: 两个字符串自左向右逐个字符相比(按 ASCII 值大小相比较),直到出现不

同的字符（区分大小写），或遇'\0'，或比较字符数超过 n 为止。当 $s1 < s2$ 时，返回为负数；当 $s1 = s2$ 时，返回值 = 0；当 $s1 > s2$ 时，返回正数。

(11) `strnicmp` - 限定长度的字符串比较（不区分大小写）

函数原型: `int strnicmp(const char *s1, const char *s2, int n);`

功能描述: 两个字符串自左向右逐个字符相比（按 ASCII 值大小相比较），直到出现不同的字符（不区分大小写），或遇'\0'，或比较字符数超过 n 为止。当 $s1 < s2$ 时，返回为负数；当 $s1 = s2$ 时，返回值 = 0；当 $s1 > s2$ 时，返回正数。

(12) `strlen` - 计算字符串长度

函数原型 1: `int strlen(const char *s);`

函数原型 2: `int strlen(string s);`

功能描述: 返回字符串长度。

(13) `memcpy` - 内存数据复制

函数原型: `void *memcpy(void *dest, void *src, int n);`

功能描述: 从存储区 `src` 复制 n 个字节到存储区 `dest`。返回指向目标存储区 `dest` 的指针。

(14) `memcmp` - 内存数据比较

函数原型: `int memcmp(const void *data1, const void *data2, int n);`

功能描述: 把存储区 `data1` 和存储区 `data2` 的前 n 个字节进行比较。如果返回值 == 0, 则表示 `data1` 等于 `data2`; 如果返回值 < 0, 则表示 `data1` 小于 `data2`; 如果返回值 > 0, 则表示 `data2` 小于 `data1`。

(15) `string` - 标准字符串构造方法

➤ 函数原型 1: `string string(int len);`

功能: 构造并返回一个预留空间长度为 len 的空字符串。

➤ 函数原型 2: `string string(void *str, int len);`

功能: 构造一个长度为 len 的空字符串，并用 `str` 指向的数据进行初始化。

➤ 函数原型 3: `string string(string1, string2, ...);`

功能: 将若干个 `string` 或 `char *` 类型的字符串依次首尾连接起来，构造出一个新字符串返回。

(16) `unix_timestamp` - 获取 32 位 unix 时间戳

函数原型: `unsigned int unix_timestamp(void);`

入口参数: 无

返回值: 返回 32 位无符号整数。

功能描述：生成 32 位 unix 时间戳，即从 1970-1-1 00:00:00 到当前的秒数。

(17) genAutoID - 生成 32 位自增流水 ID

函数原型：unsigned int genAutoID(void);

入口参数：无

返回值：返回 32 位无符号整数。

功能描述：初始值为 1，每调用一次，返回值自动加 1。

(18) random - 生成随机数/随机选择集合数据

函数原型 1: int random (int maximum);

功能描述：生成一个绝对值小于入参 maximum 的 32 位随机数。如果不指定上限（省略 maximum 参数），则随机生成 1 个 32 位随机数。

函数原型 2: var random (var1, var2, ...);

入口参数：二个以上任意类型数据

返回值：随机返回入口参数列表中的一个。

功能描述：随机选择集合数据，即从多个数据中随机选择一个数据返回。

例如：random(100, 0x255, 123.456, 'x', "abcdefg")，实现从入口参数列表中随机返回一个数据，入口参数的数据类型可以自由混合，返回值类型就是随机选择参数的实际数据类型。

(19) reverse 逆转数据的字节顺序

函数原型：var reverse(data, maxLen);

入口参数：参数 data 为待逆序的源数据，可以是整形（短整形或长整形）或浮点等基本数据类型，也可以是字节型数组、字符串或数据指针类型；可选参数 maxLen 用于指定数据转换的最大长度，如果省略该参数则转换长度取源数据 data 的默认长度值，如 int 类型数据默认长度 4 字节，short 类型默认 2 字节，字符串则自动获取字符串自身长度，等等。

功能描述：将源数据 data 的字节顺序高低逆转后返回逆序重排的数据。如果源数据类型是整形（短整形或长整形）或浮点等基本数据类型，则不会修改源数据的字节顺序，而是返回逆序后的数据；如果源数据类型是字符串、数组或指针类型，则源数据也会发生字序逆转，并返回逆序后的数据引用。

(20) gets - 从当前指令数据中复制数据段

函数原型：var gets(offset|#comment, len);

入口参数：offset|#comment 为偏移地址或者字段注解名。如果是通过偏移地址复制数据，则需要明确这个偏移地址是相对当前模板对应的指令数据；如果通过字段注解名复制数据，则系统会优先查找源指令帧对应的注解字段，如果不存在则再查找应答指令帧对应注解字段，并且引用的目标注解名必须确保在当前模板中调用 gets 函数前已经定义过。

功能描述：从指定位置（当前指令帧 offset 偏移地址，或者模板字段注解名对应的指令

数据段)处,拷贝指定长度为 len 的数据块;长度参数 len 可以省略,如果使用偏移地址拷贝,则省略长度参数时将拷贝数据直至指令帧末尾;如果使用字段名注解,则省略长度参数时表示拷贝注解名对应的整个字段的数据。

例如,以下这条应答规则:

指令匹配模板: ABCD\[2#command]

指令应答模板: ACK_[4:get(0)]_[2:get(#command)]

假定,自动应答规则引擎收到字符串数据 ABCDFF,则指令匹配成功,其中定长模糊匹配\[2#command]所匹配的内容是 FF。根据指令应答模板,生成应答数据为: ACK_ABCD_FF。其中,[4:get(0)]表示从当前收到的数据报文(ABCDFF)的偏移地址 0 开始拷贝 4 个字节数据:ABCD; [2:get(#command)]表示从当前收到的数据报文(ABCDFF)的注解#command 所匹配对应的偏移地址处拷贝 2 个字节数据:FF。

(21) getchar - 从当前指令数据中复制一个字节有符号数

函数原型: char getchar(offset|#comment);

函数别名: getS8

入口参数: 指令帧偏移地址或者模板字段注解名(参考前文 gets 函数的参数说明)。

功能描述: 从指定位置(当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处)拷贝 1 个字节的有符号数据。

(22) getuchar - 从当前指令数据中复制一个字节无符号数

函数原型: unsigned char getuchar(offset|#comment);

函数别名: getU8、getByte

入口参数: 指令帧偏移地址或者模板字段注解名(参考前文 gets 函数的参数说明)。

功能描述: 从指定位置(当前指令帧 offset 偏移地址,或者模板字段注解名对应的指令数据段处)拷贝 1 个字节的无符号数据。

(23) getshort - 从当前指令数据中复制 2 个字节有符号整数

函数原型: short getshort(offset|#comment, isBigEndian);

函数别名: getS16

入口参数: offset|#comment 为偏移地址或者字段注解名; isBigEndian 为可选参数,布尔型数据变量,表示指定读取数据的字节顺序是高字节在前还是在后。为 true 表示高字节在前否则低字节在前。如果省略 isBigEndian 参数,表示取全局的字节顺序设置(自动应答设置窗口面板右下方的“网络字序”复选框,用于设置全局的默认字节顺序,勾选表示全局字序为 BigEndian,否则为 LittleEndian)。

功能描述: 从指定位置(当前指令帧 offset 偏移地址处,或者模板字段注解名对应的指令数据段)拷贝 2 个字节的有符号整数。

(24) getushort - 从当前指令数据中复制 2 个字节无符号整数

函数原型: `unsigned short getushort(offset|#comment, isBigEndian);`

函数别名: `getU16`

入口参数: `offset|#comment` 为偏移地址或者字段注解名; `isBigEndian` 为可选参数, 布尔型数据变量, 表示指定读取数据的字节顺序是高字节在前还是在后。为 `true` 表示高字节在前否则低字节在前。如果省略 `isBigEndian` 参数, 表示取全局的字节顺序设置。

功能描述: 从指定位置 (当前指令帧 `offset` 偏移地址, 或者模板字段注解名对应的指令数据段处) 拷贝 2 个字节的无符号整数。

(25) `getint` - 从当前指令数据中复制 4 个字节有符号整数

函数原型: `short getint(offset|#comment, isBigEndian);`

函数别名: `getS32`

入口参数: `offset|#comment` 为偏移地址或者字段注解名; `isBigEndian` 为可选参数, 布尔型数据变量, 表示指定读取数据的字节顺序是高字节在前还是在后。为 `true` 表示高字节在前否则低字节在前。如果省略 `isBigEndian` 参数, 表示取全局的字节顺序设置。

功能描述: 从指定位置 (当前指令帧 `offset` 偏移地址, 或者模板字段注解名对应的指令数据段处) 拷贝 4 个字节的有符号整数。

(26) `getuint` - 从当前指令数据中复制 4 个字节无符号整数

函数原型: `unsigned short getuint(offset|#comment, isBigEndian);`

函数别名: `getU32`

入口参数: `offset|#comment` 为偏移地址或者字段注解名; `isBigEndian` 为可选参数, 布尔型数据变量, 表示指定读取数据的字节顺序是高字节在前还是在后。为 `true` 表示高字节在前否则低字节在前。如果省略 `isBigEndian` 参数, 表示取全局的字节顺序设置。

功能描述: 从指定位置 (当前指令帧 `offset` 偏移地址, 或者模板字段注解名对应的指令数据段处) 拷贝 4 个字节的无符号整数。

(27) `sizeof` - 计算数据长度

函数原型: `int sizeof(var|#comment);`

入口参数: `var|#comment` 为数据变量或者字段注解名;

功能描述: 返回目标变量或注解字段的数据长度。

版本要求: 该函数要求调试助手版本号 \geq V5.0.8.6

(28) `calculate` - 计算校验位

函数原型: `var calculate(offset, len, algorithm);`

入口参数: 参数 `offset` 为待校验数据偏移地址; `len` 为待校验数据长度, 如果 `len` 为 -1, 表示数据长度截止到当前 `calculate` 函数调用位置的前一个字节数据, `len` 为 -2 则表示数据长度从当前位置往前推 2 个字节, 以此类推; 参数 `algorithm` 为选择的算法。

功能描述: 计算数据校验值。函数返回值的数据类型根据具体的校验算法而定。如果校

验值是单字节的，则返回的数据类型是 unsigned char，如果校验值是双字节的，则返回值的数据类型是 unsigned short；如果校验值是四字节的，则返回值数据类型是 unsigned int。

目前 calculate 支持的算法（参数 algorithm 可取常量值）列表如下：

- ALGO_ACC //1 字节累加和
- ALGO_LRC //1 字节累加和再取负
- ALGO_XOR //1 字节异或校验
- ALGO_CRC16_MODBUS //2 字节 MODBUS CRC16 校验
- ALGO_CRC8
- ALGO_CRC8_ITU
- ALGO_CRC8_ROHC
- ALGO_CRC8_MAXIM
- ALGO_CRC8_WCDMA
- ALGO_CRC8_CDMA2000
- ALGO_CRC16_CCITT
- ALGO_CRC16_CCITT_FALSE
- ALGO_CRC16_XMODEM
- ALGO_CRC16_X25
- ALGO_CRC16_IBM
- ALGO_CRC16_USB
- ALGO_CRC16_MAXIM
- ALGO_CRC16_DNP
- ALGO_CRC32
- ALGO_CRC32_BZIP2
- ALGO_CRC32_MPEG2
- ALGO_CRC32_POSIX
- ALGO_CRC32_JAMCRC

(29) echo - 实现 BLOCK 代码块的流式返回值

函数原型：int echo (const char *format, ...);

功能描述：格式化输出的文本数据作为 BLOCK 代码块的最终返回值，而 echo 函数本身将返回所生成文本数据的长度。

例如：

```
\[{ echo("hello\r\n"); echo("ok\r\n") }]
\[{ echo("ERR=%d",errno); }]
\[{ echo (gets(#Annotation)); }]
```

注，echo 的第二个参数可以省略。当只有 1 个参数时，不做格式化处理，直接输出第一个参数指向的数据。形式为 \[{...}] 的 BLOCK 代码块，可以通过 return 语句或者 echo 函数来实现整个 BLOCK 的返回值。不同的是 return 语句执行后就会立即退出当前 BLOCK，不

会再执行当前 BLOCK 的后续代码；而 echo 语句则不会退出 BLOCK，后续如果有多个 echo 函数输出数据，这些数据会流式拼接在一起作为当前 BLOCK 的返回值。

(30) echob - 实现 BLOCK 代码块的流式返回值

函数原型：void echob (data, length);

入口参数：data 为输出的数据内容（可以是整形数或字符串或数组或数据指针等）；length 为限定输出数据的字节长度。

功能描述：输出二进制数据作为 BLOCK 代码块的返回值。同一个 BLOCK 内的多个 echo 或 echob 函数的输出会按流式数据的方式依次追加合并。

例如：

```
\[{ echob(gets(0),100)); }]  
\[{ echob(gets(#Annotation)); }]  
\[{ char str[3]="\x01\x02\x03"; echob(str,3); }]
```

注， echob 的第二个参数可以省略。当只有 1 个参数时，echo 和 echob 这两个函数等价，按输入数据的固有长度输出。

(31) delay - 延迟操作

函数原型：void delay(ms);

入口参数：ms 为延迟的毫秒数。

功能描述：这个函数主要用于应答模板，实现延迟应答。

例如以下应答模板：

```
\[{delay(1000); echob(gets(0),100));}]  
\[{delay(500);}]OK\r\n
```

通过在应答模板头部插入 delay 函数，实现应答数据的延迟输出(发送)。

第七章 自动应答规则设计

使用调试助手软件的“指令自动应答”功能，要预先根据应用场景建立若干条自动应答规则。当调试助手软件接收到通信数据/指令帧时，会按照其优先级顺序逐一遍历匹配自动应答规则列表，直至找到一条相匹配的应答规则，然后根据对应的应答模板自动进行应答回复。自动应答规则的具体设计规则将会在本章节中详细介绍。

7.1 应答规则概述

自动应答功能，用于实时对调试助手接收到的指令/数据进行匹配/识别，并自动按用户预定义规则发送相应的应答数据。用户只须事先设计好应答规则，然后调试助手内部集成的规则引擎会自动对接收到的数据进行指令规则匹配及应答数据的发送。如果需要同时支持多条指令的自动应答，只要相应地建立多条自动应答规则。如果某一条指令帧数据能匹配多个自动应答规则，则最终只会响应那个优先级序号最高的规则。

每一条应答规则都由一对“指令匹配模板”和“指令应答模板”构成，每个模板的数据格式可以是十六进制形式，或者是 ASCII 码字符串形式。其中，ASCII 码形式的指令模板数据，可嵌入基于类 C 语言的脚本代码，方便用户实现更加灵活、强大的自动应答功能。



图 7-1 用户创建的应答规则列表窗口以及右键菜单

自动应答规则的管理面板，见图 7-1 所示。窗口中按优先级顺序逐条显示用户定义的自动应答规则。每一条应答规则项必须勾选相应的复选框后才能启用，同时还必须勾选窗口右下方名为“启动自动应答”的总开关后才能最终开启自动应答功能。

通过自动应答窗口的右键菜单(如图 7-1 右侧所示)，可以对应答规则做各种增、删、改、查、导入、导出等操作。另外，在窗口左下角有一排图标快捷按钮，用于常规快捷操作。比如，“新建”命令用于新建一条自动应答规则；“删除”命令用于删除选中的规则项目；“上移”和“下移”就是调整自动应答规则的优先级顺序。注意，当调试助手接收到通信数据时，会按照此优先级顺序进行指令模板匹配，优先级数字越小，优先级级越高，一旦成功匹配一条规则，就会发送相应的应答数据，并停止继续往下匹配。

关于自动应答的“粘包”及“半包”问题说明。“粘包”问题就是，当发送端密集发出

多个连续的数据报文时，由于间隔太短或者通信延迟问题，有可能到达接收端时会粘合成一个整个没有时隙间隔的大包，无法分离出原来的多个小包；而“半包”问题就是当发送端发出一个较大的数据报文时，由于中继环节或本地软/硬件接收队列容量的问题，可能导致自动分包的产生，也就是最终接收到的是多个分开的小包。“粘包”及“半包”问题在串口以及 TCP 等流式通信时无法避免。注意：“自动应答”模块支持对“粘包”数据进行自动分包后再处理，但不支持对“半包”数据进行自动组包后再匹配。

7.2 应答规则入门

在自动应答管理窗口，通过右键菜单的“新建…”命令或者窗口左下角的快捷图标按钮，进入到“添加新规则”界面，如下图所示。

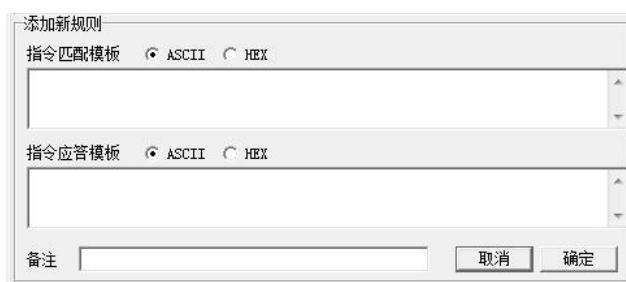


图 7-2 添加新规则编辑窗口

每一条应答规则都包含“指令匹配模板”及“指令应答模板”这两个必填的数据输入项，前者用于对接收的指令数据进行特征匹配识别，后者用于提供自动应答的数据帧；“备注”为选填的注释性文字，便于用户辨识或理解。最后点击“确认”按钮进行保存。“指令匹配模板”及“指令应答模板”的数据可以选择 ASCII 码格式或十六进制（HEX）形式，而最终选择的格式必须跟实际输入的数据一致。下面列举几个简单的例子来帮助理解。

7.2.1 ASCII 码应答规则创建实例

一个简单的示例，如下图 7-3(a)：指令匹配模板输入 ASCII 码字符串“how are you”，指令应答模板输入 ASCII 码字符串“I’m fine.”，备注项随便填，比如“greet”。最后点击“确定”按钮保存。如图 7-3(b)所示，在自动应答模块窗口就会出现新建的规则项。



图 7-3(a) 新建 ASCII 码应答规则示例



图 7-3(b) 应答规则列表管理窗口

如果要进行自动应答的测试，只须按照如图 7-3(b)所示，勾选目标应答规则项前面的复选框，并勾选（点亮）右下角的总开关“启动自动应答”。此时，只要接收到包含“how are you”的数据帧，就会自动回复应答报文帧“I’m fine.”。

7.2.2 十六进制应答规则创建实例

如果指令模板中要求包含非打印的 ASCII 码字符，无法直接输入时，可以选择十六进制模式下输入十六进制数据的方式实现，或者在 ASCII 码模式下使用转义字符的方式实现。示例如下图所示：

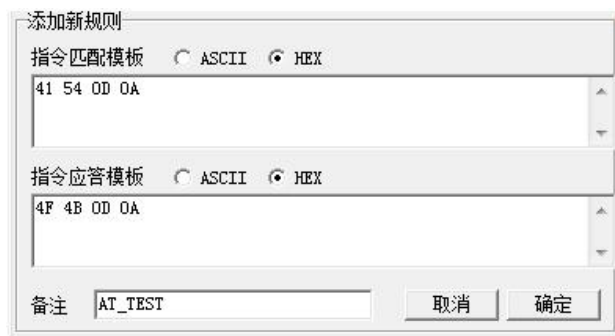


图 7-4 新建 HEX 码应答规则示例

实际上，此例创建的是一条 AT 指令。等价地，可以使用包含 16 进制转义符的 ASCII 码字符串来表示，如下图所示。



图 7-5 新建包含 16 进制转义符的 ASCII 码应答规则示例

补充说明，本例所描述的 AT 指令应答规则，还可以采用字符串混合转义字符的方式实现。如下图所示：



图 7-6 新建字符串混合转义字符应答规则示例

注意，回车换行符除了可以用转义 16 进制 ASCII 码 \x0D\x0A 的形式表示外，还可以用 \r\n 表示。



图 7-7 应答规则列表管理窗口

以上图 7-4、7-5、7-6 所创建三条规则 AT_TEST、AT_TEST2、AT_TEST3 相互等价，其实现的效果相同，都是在接收到 AT 指令“AT\r\n”后发送应答数据“OK\r\n”。

实际应用时，“指令匹配模板”和“指令应答模板”的编码形式（ASCII/HEX）可根据实际的应用场景灵活选择：可以都为 ASCII 码或都为 HEX 码，也可以一个为 ASCII 码，另一个为 HEX 码。但要注意的是：转义字符只允许在 ASCII 码模式下使用。如果选择了十六进制模式，就不支持输入转义字符。

7.2.3 模糊匹配实现 ECHO 功能实例

所谓 ECHO 功能，就是收到任意数据帧内容，都原样发送回去。在介绍如何实现 ECHO 功能前，需要明确一点：自动应答规则触发的前提是指令匹配，而指令匹配方式是通过指令匹配模板来定义的。指令匹配模板支持使用通配符进行模糊匹配，从而可实现灵活的指令匹配方式。

模糊匹配使用的通配符有两种：一种是弹性模糊匹配 \[*]，该通配符用于匹配任意长度的连续数据，单独使用 \[*] 时呈贪婪匹配特性；另一种是定长模糊匹配 \[n]，其中 n 是正整数，该通配符用于匹配指定字节长度为 n 的任意连续数据。注意，通配符号是反斜杠开头的转义字符形式。

实现 ECHO 功能很简单，只要将指令匹配模板和指令应答模板都设为字符串 \[*] 即可。



图 7-8 经典 Echo 功能实现

上图所述的规则生效后，不管收到任何数据都会匹配成功，并且会将接收到的数据原样回复给对方。

7.2.4 嵌入脚本代码的高级应答规则

自动应答规则数据中可以嵌入基于类 C 语言语法的脚本代码，通过设计业务逻辑代码，允许用户在自动应答过程中，对接收的指令/数据进行判断、计算、校验及多种模式的匹配，并在应答数据中插入对源指令数据的引用以及通过调用系统函数以及运算表达式动态生成满足用户设计需求的应答数据帧。

下面通过一个简单的例子，来直观地展示下高级应答规则的基本能力。如下图所示，这是 Modbus-RTU 的 04 号命令读寄存器数据的自动应答规则示例。

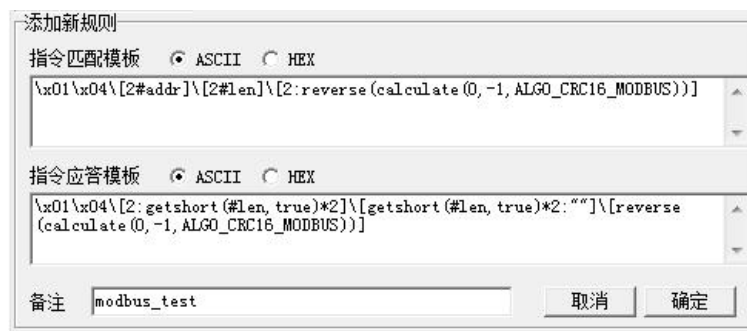


图 7-9 Modbus-RTU 读寄存器自动应答功实现

指令匹配模板中，\[2#addr]表示定长模糊匹配任意两个字节，这里用于匹配 2 个字节的地址数据；井号后面是注解，属于于注释性文字；同理，\[2#len]表示模糊匹配 2 个字节的长度数据；接下来的\[2:reverse(0,-1,ALGO_CRC16_MODBUS)]表示定长匹配两个字节的的数据，该数据的值要求等于冒号后面的计算表达式。其中，系统函数 reverse 的作用是将整形数据的高低字节交换；系统函数 calculate 用于计算校验值，其第一个参数，用于表示校验数据的偏移地址为 0，即从第一个字节开始计算校验。第二个参数为校验数据的长度，这里的-1 表示校验数据的长度截止到当前函数调用位置的前一个字节数据。第三个参数是 ALGO_CRC16_MODBUS，表示检验采用的算法。

同理，指令应答模板也采用表达式及函数的形式产生应答帧的数据内容。具体的语法规则及详细的函数定义将在后面再详细介绍。

7.3 指令匹配模板

指令匹配模板是实现自动应答的前提基础，是指令匹配的依据。每一个指令匹配模板，都由若干个指令匹配字段组成；每一个指令匹配字段的匹配模式，根据语法形式不同，可分为立即数匹配和非立即数匹配，如下图所示。



图 7-10 指令字段匹配模式分类

其中，“立即数匹配”，本质上是常量匹配，而与之相反的“非立即数匹配”，就是变量匹配。“非立即数匹配”，也称作“模式匹配”，这是因为其语法规则使用模式符\[...\]来标记具体的匹配特征。根据匹配的数据对象的不同，又可分为精确匹配和模糊匹配两种基本类型。其中，精确匹配可细分为立即数（精确）匹配和定长精确匹配两个子类型；模糊匹配又可分为定长模糊匹配及不定长（弹性）模糊匹配两个子类型。

在同一个指令匹配模板中，多种模式的匹配字段可以根据实际情况自由灵活地混合使用。

7.3.1 立即数匹配

“立即数匹配”，属于指令匹配模板的一种精确匹配方式，其所匹配的对象是固定的常数立即数，而不是变量、也不是计算表达式或其他需要动态计算的值。比如，以下几个指令匹配字段都是立即数匹配模式：

立即数字段匹配模板	模板编码
AT\r\n	ASCII
\x41\x54\x0D\x0A	ASCII
41 54 0D 0A	HEX

从语法特征来看，立即数匹配模式不能使用模式符\[...\]，因而不支持任何动态语法特性，不支持任何脚本代码，仅能使用纯粹的 ASCII 码字符串（支持转义字符）或十六进制编码常数。不管是 ASCII 编码还是十六进制编码的指令模板，都可以使用立即数匹配模式。不同的是，十六进制编码的指令模板由于不能使用模式符\[...\]，因而仅可使用立即数匹配，而 ASCII 编码的指令模板，可以混合使用多种不同的匹配模式。

7.3.2 定长精确匹配

定长精确匹配，用于精确匹配一个指定长度和内容的指令数据段。定长精确匹配的语法形式如下所示：

\[exp_len: exp_value #comment]

其中，参数 exp_len 表示匹配的数据长度，参数 exp_value 表示匹配的数据内容。exp_len 和 exp_value 在形式上都可以是常数、或运算表达式、或 BLOCK 代码块，可以在指令匹配阶段动态计算待匹配的数据长度及内容。如果 exp_len 跟 exp_value 的实际数据内容长度不一致，则最终的匹配长度以 exp_len 为准，超出则截尾，不足则补零；comment 是注解，也就是注释性文字，可以省略。在语法规则上要注意，模式符\[...\]的反斜杠跟左中括号之间没有空格、数据长度 exp_len 跟变量数据 exp_value 之间通过冒号分隔；注解 comment 以井号开头。

比如，以下几个指令匹配段都是定长精确匹配：

匹配模板	模板数据
\[2: calculate(0, -1, CRC16_MODBUS)]	调用内建函数计算匹配一个 2 字节长度的 CRC 校验位

<code>\[1: 2*(getuchar(0)+getuchar(1))]</code>	通过运算表达式计算匹配变量值
<code>\[(getuchar(#len)+1): 2*(getuchar(0)+getuchar(1))]</code>	长度及数据都是计算表达式
<code>\[{ return getint(0)}:{ byte value=getbyte(#value); return value?1:0; }]</code>	匹配数据长度及内容都是包含在 {} 内的 BLOCK 代码块, 通过 return 或 echo 返回 BLOCK 代码块的计算值

注意：定长精确匹配模式必须指定变量数据的长度及内容，两者都不可省略，仅注解是可以省略的可选项。如果省略了 exp_len 或 exp_value, 仅剩下一个参数段就变成了定长模糊匹配模式，而不再是定长精确匹配。

如果 exp_len 或 exp_value 的表达式有多条语句组成，可以通过包含在大括号对 {} 内的 BLOCK 代码块来调用，一个 BLOCK 内多条语句之间用分号隔开，BLOCK 的最终返回值通过 return 或者 echo 语句实现，例如：

```
\[ {
    int i=getuint(0);
    return i;
}:{
    if(getuchar(4)==0) return 10;
    else return 20;
}
```

7.3.3 定长模糊匹配

所谓定长模糊匹配，就是在指令的部分字段匹配时，只匹配数据长度而不关心数据的内容，也就是匹配指定长度的任意数据。其语法形式如下所示：

```
\[ exp #comment]
```

其中，参数 exp 表示匹配的数据长度，形式上可以是常数立即数或运算表达式或 BLOCK 代码块；#comment 是注解，也就是注释性文字，可以省略。在语法形式上，定长模糊匹配比定长精确匹配模式少一个用于匹配内容的参数字段。从这点也可以看出，定长模糊匹配只匹配数据长度，不匹配数据内容。

模糊匹配模式举例：

- `\[n #定长模糊匹配 n 个字节任意数据, 其中 n 为任意非零正整数]`
- `\[1 #定长模糊匹配 1 个字节任意数据]`
- `\[4 #定长模糊匹配 4 个字节任意数据]`
- `\[getint(0) #定长模糊匹配的长度为表达式计算值]`
- `\[{byte len=getbyte(0);return len;} #定长模糊匹配的长度为 BLOCK 代码返回值]`

7.3.4 弹性模糊匹配

弹性模糊匹配，简称弹性匹配，是一种不定长度的模糊匹配，支持正则字符集匹配规则，

并可设定最小及最大匹配长度,而最终实际匹配的数据长度会根据上下文数据内容以及设定的边界条件弹性自适应。弹性匹配的语法形式如下所示:

`\[*(minLen,maxLen,filter) #comment]`

上式中,星号表示弹性匹配,而星号后面的圆括号内指定了弹性匹配的三个参数(下文统称为弹性参数):弹性下界 minLen、弹性上界 maxLen、弹性过滤器 filter。如果将部分弹性参数省略掉,再将注解省略掉,则弹性模糊匹配的表达式由简至繁,共可化为四种样式:

- ① `\[*]` *#弹性参数为空,表示任意模糊匹配*
- ② `\[*(minLen)]` *#仅指定弹性下界的弹性匹配*
- ③ `\[*(minLen,maxLen)]` *#指定弹性上下界但无过滤器的弹性匹配*
- ④ `\[*(minLen,maxLen,filter)]` *#指定所有弹性参数的弹性匹配*

弹性匹配过程中,会从指定的下界(最小长度 minLen)开始,根据数据过滤规则(filter)遍历目标数据,一旦找到满足整个指令匹配模板的最小模糊匹配长度,或者匹配长度超过设定的上界(最大长度 maxLen),就会停止匹配。特别注意,一条指令模板可以包含多个弹性匹配表达式,任何位于模板模板的头部和中间的弹性匹配都是非贪婪匹配,只有位于指令模板结尾的弹性匹配才是贪婪匹配。如果某个弹性匹配既在模板头部也在尾部,也即是说这个指令模板就是有 1 个弹性匹配构成的,那么这个弹性匹配属于贪婪匹配。

- 弹性下界 minLen 参数,表示最小匹配数据长度,缺省为 0 时表示不限定最小长度。
- 弹性上界 maxLen 参数,表示最大匹配数据长度,缺省为 0 时表示不限定最大长度。
- 弹性过滤器 filter 参数,表示数据匹配规则,是中括号引用表示的正则字符集,比如 `[\d]`、`[\s]`、`[0-9A-B]` 等等。作为简化,当[正则字符集]的首尾没有空格符时,外部的括号可以省略。当 filter 参数为空或缺省时,表示不限定数据内容。

弹性过滤器对应的正则字符集,具体有如下两种形式:

- ① `[xyz]` 表示包含字符集,即匹配括号间的任何一个字符。例如, `[0-9]` 表示匹配任一数字, `[0-9a-zA-Z]` 表示匹配任一大小写字母或数字;
- ② `[^xyz]` 表示排除字符集,即匹配不在括号间的任何一个字符。例如, `[^0-9a-zA-Z]` 匹配除字母及数字外的其它任何字符。

正则字符集支持各种正则转义字符:

- `\d` 匹配一个数字字符。等同于 `[0-9]`;
- `\D` 匹配一个非数字字符,等同于 `[^0-9]`;
- `\f` 匹配一个换页符;
- `\n` 匹配一个换行符;
- `\r` 匹配一个回车符;
- `\t` 匹配一个制表符 (Tab 符);
- `\p` 匹配 CR/LF (等同于 `\r\n`);
- `\s` 匹配任何空白符,包含空格、制表符、回车换行等,等价于 `[\f\n\r\t\v]`;
- `\S` 匹配任何非空白区域,等价于 `[^\f\n\r\t\v]`;
- `\v` 匹配一个垂直制表符;
- `\w` 匹配任何包含下划线的词语,等价于 `[A-Za-z0-9_]`;

\W 匹配任何非单词字符，等价于 $[\text{^A-Za-z0-9_}]$ ；

\x 引导 16 进制数对应的字符，例如 “\x64” 表示 0x64, 即字符 “d”

实际应用场景中，一条指令模板中可以包含多个模糊匹配模式(包括定长模糊匹配和弹性模糊匹配)。例如，以下这条指令匹配模板：

`\[*]AB\[2]CD\[*]EF\[*(1,8,[\d])]GH\[3]XY\[*]`

注意：指令匹配模板的首/尾两端的无弹性参数的弹性模糊匹配都是可以省略的，这是因为自动应答规则引擎在指令匹配过程中是基于流式数据的全局匹配，其首尾会自动隐式添加弹性模糊匹配，实现全局的模糊匹配。因此，上条模板可以简化为：

`AB\[2]CD\[*]EF\[*(1,8,[\d])]GH\[3]XY`

这条指令匹配模板采用了多种匹配模式，其中 AB CD EF GH XY 这些都是立即数匹配，\[2]和\[3]为定长模糊匹配，\[*]和\[*(1,8,[\d])]为弹性模糊匹配。

【版本要求】弹性模糊匹配的弹性参数为新特性，要求调试助手软件升级到 V5.0.9 以上版本。

7.3.5 指令匹配模板示例

在实际工程应用中，各种接口协议的指令帧格式，通常都包含内容固定不变的部分以及动态变化的部分。在设计指令匹配规则时，指令中固定不变的部分通常使用“立即数匹配”，而动态变化的部分则可使用各种模式匹配，诸如定长精确匹配、定长模糊匹配、弹性模糊匹配，或结合多种匹配方式灵活运用。也就是说，实际工程应用中的指令匹配模板通常都是混合模式的，同一个指令匹配模板可同时包含多种匹配模式。

例如，下面这一条 Modbus-RTU 的 04 号功能读寄存器的指令匹配模板：

`\x01\x04\[2#addr]\[2#len]\[2:reverse(0,-1,ALGO_CRC16_MODBUS))]`

该指令匹配模板字段分解如下表：

序号	字段	匹配模式	描述
1	\x01	立即数匹配	匹配设备 ID
2	\x04	立即数匹配	匹配 Modbus 功能号
3	\[2#addr]	定长模糊匹配	匹配 2 字节的寄存器地址
4	\[2#len]	定长模糊匹配	匹配 2 字节的数据读取长度
5	\[2:reverse(0,-1,ALGO_CRC16_MODBUS))]	定长精确匹配	匹配 2 字节的 CRC16 校验码

7.4 指令应答模板

指令应答模板用于生成自动应答的报文数据，是自动应答规则的两个基本组成元素之一。每个指令应答模板通常都是由若干个数据段构成，每个数据段按照其生成应答数据的方式不同，可以划分为立即数应答段以及非立即数应答段两种类型。

7.4.1 立即数应答

指令应答模板中的“立即数应答”字段，类似于指令匹配模板中的“立即数匹配”字段，其外在表现形式为不使用模式符\[]的数据段。不同的是，前者是用于生成应答数据，而后

者是用于指令匹配比较。

立即数应答数据段，示例如下：

- ASCII: OK\r\n
- ASCII: \x4F\x4B\x0D\x0A
- HEX: 4F 4B 0D 0A

从外在特征来看，立即数应答字段只使用纯粹的 ASCII 码字符串或十六进制编码数据，而不借助模式符\[]，即不通过脚本语法规则来生成应答数据段。

7.4.2 非立即数应答

所谓“非立即数应答”字段，就是排除“立即数应答”字段外的所有其他应答字段，用于生成一个指定数据长度及内容的应答数据段。“非立即数应答”，也称作“模式应答”，这是因为其外在形式表现为使用模式符\[…]，引用语法规则来生成应答数据。其一般语法形式如下所示：

```
\[ exp_len: exp_value #comment]
```

其中，参数 exp_len 表示生成字段的数据字节长度，参数 exp_value 表示生成字段的数据内容，#comment 是注解，也就是注释性文字，可省略。值得注意的是，exp_len 或 exp_value 可以是常数、或者变量，或者逻辑运算式，或者是由多语句组成的具有返回值的 BLOCK 代码块。在语法形式上，类似于指令匹配模板中的定长精确匹配语法格式，都有两个参数段，但两者用途不同，一个用于生成应答数据，而另一个则用于指令匹配比较。

参数 exp_value，不管其表现形式是立即数常量、变量还是运算表达式的计算值或者是 BLOCK 代码块，其本身就具有一个默认的数据长度。比如，int 型数据内容默认长度为 4 字节，short 型数据内容默认 2 字节。再比如，字符串数据，其本身就包含长度信息，等等。如果这个默认长度跟前置的 exp_len 值不等，则最终输出的数据长度以 exp_len 值为准，如果 exp_len 小于数据内容的默认长度，则输出数据将被截尾，否则尾部以零补足。

参数 exp_len 可以省略，也就是不进行显式指定，进而简化为如下所示的形式：

```
\[ exp #comment]
```

简化后只包含一个表达式，这个形式类似于指令匹配模板中的定长模糊匹配。区别是，定长模糊匹配模式中的单个表达式是用于计算数据长度，而格式化应答字段中的单个表达式是用于计算生成应答数据内容。最终，单表达式模式应答字段的输出长度为 exp_value 解析值的默认长度，比如 int 类型整数默认 4 字节、short 类型默认 2 字节，等等。

非立即数应答数据段，若干示例如下：

- 1) \[4:genAutoID()#生成 4 字节的 32 位流水自增 ID]
- 2) \[8:get(0) #从源指令帧的偏移地址 0 开始截取 8 个字节数据]
- 3) \[2: getushort(2)*2 #从请求指令帧偏移地址 2 处取一个 16 位整数后乘 2]
- 4) \[2:random() #返回一个 32 位随机数，并截取 2 个有效字节]
- 5) \[random("ABCDE",0xF1F2,1024, 'x') #从参数列表中随机返回一个数据]
- 6) \[{int i=getint(0);return i;}:{if(getbyte(5)) return 1;else return 0;} #
数据长度和内容都使用 BLOCK 代码块]

注：自动应答规则的指令模板可以嵌入类 C 语言语法的脚本代码，方便用户灵活地编写指令模板，实现强大的自动应答功能。详细的语法规则参考第六章内容。

7.4.3 指令应答模板示例

实际应用场景中，一个典型的指令应答模板，通常都会混合多个立即数应答数据段和非立即数应答数据段。这里，仍然用 Modbus-RTU 读寄存器数据的自动应答规则为例：

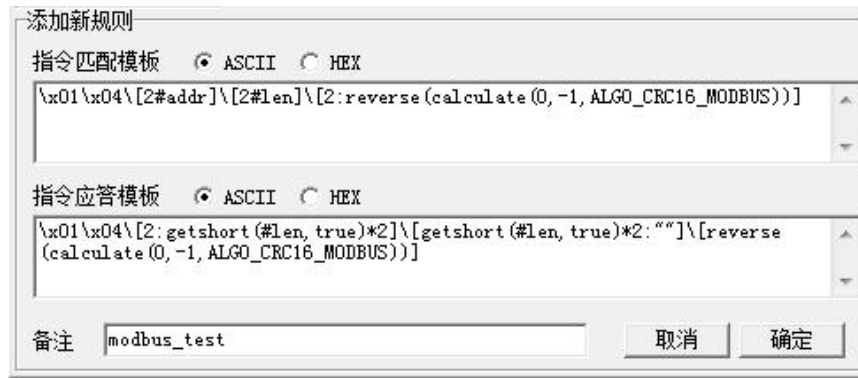


图 7-11 Modbus-RTU 的 04 号指令自动应答

这里重点关注指令应答模板数据，其各字段分解如下表：

序号	字段	应答模式	描述
1	\x01	立即数应答	设备 ID
2	\x04	立即数应答	功能号
3	\[2:getshort(#len, true)*2]	非立即数应答	2 个字节长度的数据，表示读取的寄存器数据的字节长度。其中，getshort(#len, true) 表示从请求指令中读取一个位于 #len 注解字段的 16 位整数。源指令中的长度是 16 位寄存器的长度，换算成字节长度要乘 2。
4	\[getshort(#len, true)*2:“”]	非立即数应答	生成长度为 getshort(#len, true)*2 的空数据
5	\[2:reverse(0,-1,ALGO_CRC16_MODBUS)]	非立即数应答	计算生成 2 个字节的 CRC16 校验码。函数 calculate 用于校验位计算，reverse 用于高低字节交换，以符合 Modbus-RTU 协议规范。

7.5 应答规则设计实例

7.5.1 实现指令帧 ECHO 功能

指令帧 ECHO 功能，就是收到任何指令帧数据，都原样发送回去。可按如下方式建立自动

应答规则：

✧ 指令匹配模板<ASCII>: \[*]

✧ 指令应答模板<ASCII>: \[*]

指令匹配模板使用\[*]，即弹性模糊匹配模式，不管调试助手收到任何报文数据，都能成功匹配。

指令应答模板使用\[*]，表示应答数据直接使用所匹配成功的整个源指令报文。

注意，通配符\[*]在指令匹配模板中是指弹性模糊匹配，而在应答模板中出现时，实际上是\[gets(0)]的语法糖。内建系统函数 gets，用于从源指令报文的指定偏移量开始复制数据直至报文尾部，这里参数 0 表示从头开始复制源指令报文，不指定复制的长度就是默认复制整个指令帧数据。

7.5.2 模拟 AT 指令注册网络

✧ 命令格式<ASCII>: AT+CREG=mode<CR>

✧ 命令返回<ASCII>: OK<CR> 或者 ERROR<CR>

AT 命令中的 mode 的值共有三个选项，分别是 0 or 1 or 2，如果指令操作成功将返回 OK<CR>，否则返回 ERROR<CR>

下面根据此案例来建立应答规则。由于指令模拟无法确定 AT 指令是否执行成功，只假定参数正确便能执行成功。设计应答规则如下：

✧ 指令匹配模板<ASCII>: AT+CREG=[*(1,1,\d) #mode]\r

指令匹配模板字段分解如下：

序号	字段	匹配模式	描述
1	AT+CREG=	立即数匹配	匹配固定的指令头
2	\[(1,1,\d) #mode]	弹性模糊匹配	正则匹配一个 1 个数字符，字段注解名为#model
3	\r	立即数匹配	匹配一个回车符

✧ 指令应答模板<ASCII>:

```
\[{ int mode= atoi(gets(#mode));
    return (mode>=0 && mode<=2)?"OK":"ERROR";
}]\r
```

以上指令应答模板中，gets(#mode)表示从接收到的指令数据中读取注解字段#mode 的数据内容，然后通过 atoi 函数将字符串类型转换成整型数据，再进而判断，如果该 mode 值在 0~2 范围内则返回 OK，否则返回 ERROR。

启用此条自动应答规则后，调试助手如果收到 ASCII 码指令 AT+CREG=1<CR>，则自动生成 ASCII 码应答数据 OK，如果收到 ASCII 码指令 AT+CREG=3<CR>，则自动生成 ASCII 码应答数据 ERROR。

7.5.3 模拟 AT 指令检查网络信号强度

● 命令格式<ASCII>: AT+CSQ<CR>

- 命令返回<ASCII>: +CSQ: rssi,ber<CR>

其中, rssi 为信号强度值, 应在 0 到 31 之间, ber 为误码率, 值在 0 到 99 之间。

该案例的自动应答规则, 可以按如下方式实现:

✧ 指令匹配模板: AT+CSQ\r

✧ 指令应答模板: +CSQ: \[IntToStr(random(31))],0\r

或者: +CSQ: \[{int rssi= random(31);echo("%d",rssi);}],0\r

当调试助手收到 AT+CSQ<CR>时, 指令匹配成功, 并自动发送命令应答数据。本例中, 应答指令中的信号强度值 rssi 通过系统函数 random(31) 取 0~31 的随机数, 误码率 ber 直接固定为 0。由于 AT 指令是文本命令, 所以随机生成的信号强度值须通过内建系统函数 IntToStr 将整型数据转换成字符串类型, 或者通过 echo 函数格式化成字符串类型来作为 BLOCK 代码块的返回值。

7.5.4 模拟 AT 指令交互读写通信模块参数

- AT 写指令 (配置通信模块的短信模式):

➢ AT 命令格式<ASCII>: AT+CMGF=<mode><CR>

➢ 命令返回<ASCII>: +CMGF: OK<CR>

- AT 读指令 (读取通信模块的短信模式):

➢ AT 命令格式<ASCII>: AT+CMGF?<CR>

➢ 命令返回<ASCII>: +CMGF: <mode><CR>OK<CR>

要求模拟实现配置的短信模式能够保存在内存中, 等收到读取指令时, 自动按上一次成功配置的数据来进行应答。这就要求分别实现读/写两条应答规则, 并且通过全局变量保存中间数据 (短信模式), 实现多条之间的上下文传参。

- 写指令应答规则:

➢ 指令请求模板 AT+CMGF=\[* #mode]\r

➢ 指令应答模板 +CMGF: OK\r\[{global["sms_mode"]=gets(#mode);}]

- 读指令应答规则:

➢ 指令请求模板 AT+CMGF?\r

➢ 指令应答模板 +CMGF:\[global["sms_mode"]]\rOK\r

说明: 在 AT 写指令应答规则中, 将匹配到的 mode 值保存到全局变量 sms_mode 中; 然后在 AT 读指令应答规则中, 将全局变量 sms_mode 作为模式值返回。

7.5.5 模拟 Modbus-RTU 读取寄存器数据

✧ 命令请求格式: 1 字节设备 ID | 1 字节功能号 | 2 字节寄存器地址 | 2 字节寄存器数量 | 2 字节 CRC 校验

✧ 命令返回格式: 1 字节设备 ID | 1 字节功能号 | 2 字节读取数据长度 | 寄存器数据内容 | 2 字节 CRC 校验

这里假定设备 ID 为 1, 功能号为 4, 若要求模拟 Modbus-RTU 读取寄存器数据时, 自动返回随机的寄存器数据。该案例的自动应答规则, 可以按如下方式实现:

✧ 指令匹配模板：

```
\x01\x04\[2#addr]\[2#quantity]\[2:reverse( calculate(0,-1,ALGO_CRC16_MODBUS))]
```

➤ 指令匹配模板字段分解如下：

序号	字段内容	匹配模式	描述
1	\x01	立即数匹配	精确匹配 1 个字节设备 ID
2	\x04	立即数匹配	精确匹配 1 个字节设备 ID
3	\[2#addr]	定长模糊匹配	模糊匹配 2 个字节的寄存器地址，并设置字段注解#addr
4	\[2#quantity]	定长模糊匹配	模糊匹配 2 个字节的寄存器数量，并设置字段注解#quantity
5	\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))]	定长精确匹配	精确匹配 2 字节长度的 CRC16 校验码

✧ 指令应答模板：

```
\x01\x04\[2:getshort(#quantity)*2]\[getshort(#quantity)*2:""]\[2:reverse( calculate(0,-1,ALGO_CRC16_MODBUS))]
```

其中，\x01\x04 为立即数应答字段，\[2:getshort(#quantity)*2] 为 2 字节的数据长度，其中 getshort(#quantity) 表示从源指令中获取寄存器数量，由于是 16 位寄存器，所以要乘以 2 才能表示数据的字节长度；\[getshort(#quantity)*2:""] 表示生成长度为 getshort(#quantity)*2 的空数据；\[2:reverse(calculate(0,-1,ALGO_CRC16_MODBUS))] 表示生成 2 字节的 CRC16 校验码，其中 calculate 用于计算校验位，reverse 函数用于逆转数据的高低字节顺序，以满足指令协议要求。

7.5.6 模拟 Modbus-TCP 交互读写单个寄存器

本实例要求实现两组指令模板：第一组是写寄存器指令，写入的数据保存在全局变量中；第二组是读寄存器指令，将保存在全局变量中的寄存器模拟数据读出来返回给用户。

➤ 写单个寄存器的指令匹配模板：

序号	字段内容	匹配模式	字段描述
1	\[2 #syncid]	定长模糊匹配	匹配 2 字节指令流水 ID
2	\x00\x00	立即数匹配	ModbusTCP 协议标识位
3	\x00\x06	立即数匹配	匹配接下来数据的长度
4	\[1 #deviceID]	定长模糊匹配	1 字节设备 ID
5	\x06	立即数匹配	功能码 06(表示写单个寄存器)
6	\[2 #RegAddr]	定长模糊匹配	2 字节的寄存器地址
7	\[2 #RegValue]	定长模糊匹配	2 字节的寄存器内容

➤ 写单个寄存器的指令应答模板：

序号	字段内容	匹配模式	字段描述
1	\[*]	模式应答	原样复制整个请求指令作为应答数据
2	\[{ String regName="reg"+getshort(#RegAddr, true);	模式应答	BLOCK 代码块，无应答数据产生。实现将写入寄存器的数据保存到全局变量中。

	<code>global[regName]=gets(#RegValue);</code> <code>}]</code>		
--	--	--	--

➤ 读单个寄存器的指令匹配模板:

序号	字段内容	匹配模式	字段描述
1	<code>\[2 #syncid]</code>	定长模糊匹配	2 字节指令流水 ID
2	<code>\x00\x00</code>	立即数匹配	ModbusTCP 协议标识位
3	<code>\x00\x06</code>	立即数匹配	负载数据长度(接下来数据的长度)
4	<code>\[1 #deviceID]</code>	定长模糊匹配	1 字节设备 ID
5	<code>\x04</code>	立即数匹配	功能码 04 (表示读输入寄存器)
6	<code>\[2 #RegAddr]</code>	定长模糊匹配	2 字节的寄存器地址
7	<code>\x00\x02</code>	立即数匹配	数据读取长度 (16 位寄存器 2 字节)

➤ 读单个寄存器的指令应答模板:

序号	字段内容	匹配模式	字段描述
1	<code>\[2:gets(#syncid)]</code>	模式应答	原样复制请求指令中流水 ID
2	<code>\x00\x00</code>	立即数应答	ModbusTCP 协议标识位
3	<code>\x00\x05</code>	立即数应答	负载数据长度(接下来数据的长度)
4	<code>\[1:gets(#deviceID)]</code>	模式应答	1 字节设备 ID
5	<code>\x04</code>	立即数应答	功能码 04(表示读输入寄存器)
6	<code>\x02</code>	立即数应答	读取的字节数 (单个寄存器长 2 字节)
7	<code>\[2: { String regName="reg"+getshort(#RegAddr, t rue); return global[regName]; }]</code>	模式应答	取自全局变量中保存的寄存器值

7.5.7 模拟部标 808 协议设备鉴权指令

指令匹配模板:

序号	字段内容	匹配模式	字段描述
1	<code>\x7e</code>	立即数匹配	1 字节指令头标识位
2	<code>\x01\x02</code>	立即数匹配	2 字节消息 ID
3	<code>\x00\x06</code>	立即数匹配	2 字节消息属性
4	<code>\[6 #mobile]</code>	定长模糊匹配	6 字节手机号 BCD 码
5	<code>\[2 #syncid]</code>	定长模糊匹配	2 字节指令流水 ID
6	<code>\[*]</code>	弹性模糊匹配	匹配鉴权码信息
7	<code>\[1:calculate(1, -1, ALGO_XOR)]</code>	定长精确匹配	1 字节异或校验位
8	<code>\x7e</code>	立即数匹配	1 字节指令尾标识位

指令应答模板 (鉴权通用应答)

序号	字段内容	匹配模式	字段描述
1	<code>\x7e</code>	立即数应答	1 字节指令头标识位
2	<code>\x80\x01</code>	立即数应答	2 字节消息 ID
3	<code>\x00\x05</code>	立即数应答	2 字节消息属性

4	\[6:gets(#mobile)]	立即数应答	6 字节手机号 BCD 码
5	\[2:genAutoID()]	非立即数应答	2 字节当前指令流水 ID
6	\[2:gets(#syncid)]	非立即数应答	2 字节请求指令流水 ID, 此流水 ID 要跟原请求指令的流水 ID 号一致
7	\[2:gets(1,2)]	非立即数应答	复制原指令的消息 ID
8	\x00	立即数应答	1 字节错误码
9	\[1:calculate(1,-1,ALGO_XOR)]	非立即数应答	1 字节异或校验位
109	\x7e	立即数应答	1 字节指令尾标识位

7.5.8 指令模板 debug 调试方法示例

设计指令模板时,如果无法正确地按设计意图实现指令匹配或者无法生成正确的应答数据段,可以在指令模板中插入调试打印语句来检查指令模板中各个变量字段的实际数据值,通过逐一排查比对的方式来定位错误故障点。

不管是指令匹配模板,还是指令应答模板,都可以插入形如\[{...}\]的 BLOCK 代码块,只要这个 BLOCK 代码块内没有调用 return 语句或者 echo/echob 函数来生成返回值,那么这个 BLOCK 代码块就不会对指令匹配或者指令应答起任何作用。但是,我们可以在 BLOCK 代码块中插入 printf 函数,将关键的匹配字段或者应答字段对应的数据或变量打印出来,尤其要检查指令匹配模板中定长精确匹配字段的对应的数据,通过比对来排查错误。

例如,假定有一指令模板如下所示:

- 指令匹配模板 \xFF\[4 #Payload]\[1: calculate(0,-1,ALGO_ACC))]
- 指令应答模板 \xFF\x00

通信测试实验中,当自动应答规则引擎接收到十六进制数据 FF 00 00 00 00 00 时,如果没有发生应答动作。那么我们就要对指令匹配模板进行检查,主要方法是将“定长精确匹配”字段的值打印出来进行比对,并且必须在匹配字段前进行打印。这是因为,如果前面的字段匹配失败则后面的模式代码都不会被执行到。因此,可以修改指令匹配模板(插入 BLOCK 调试代码块)如下:

```
\xFF\[4#Payload]\[ {printf("checksum==%02x",calculate(0,-1,ALGO_ACC));}]\[1:calculate(0,-1,ALGO_ACC))]
```

然后再次接收数据 FF 00 00 00 00 00 时,调试助手打印信息 checksum==FF,而实际接收到的数据帧的校验位是 00,从而可以判定问题出在校验位的匹配上,再进一步检查用于校验位计算匹配的代码,最终发现是校验数据的偏移地址错误,应将指令匹配模板修改为:

```
xFF\[4 #Payload]\[1: calculate(1,-1,ALGO_ACC))]
```

通过插入 BLOCK 级的调试代码,有利于缩小故障点的排查范围,帮助用户快速地定位问题和解决问题。