USER MANUAL

Revision 4
Nov-02-2022

用户手册

版本4
2022年11月02日

Button Commands

MODE	COMMAND KEYS				OPCODE	OPER X	OPER Y
DIR	**Carry**	**Save**	**Load**	**Clock**	**Opcode**	**Oper X**	**Oper Y**
	Toggle Carry Flag	Send Program Memory to Serial Port	Load Program Memory from Serial Port	Master Clock source	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)
ALT		Save Program Memory to selected Flash	Load Program Memory from selected Flash		Dimmer level select	Baud Rate select	Flash portion select for Save / Load
SS	**Historry**	**- Addr**	**Addr +**	**Step**	**Opcode**	**Oper X**	**Oper Y**
	Enter History submode	Decrement Program Memory Address	Increment Program Memory Address	Execute one instruction	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)
ALT	Toggle Carry Flag	Reset Program Memory Address to 0x000	Preset Program Memory Address to the last word used	Address set from Opcode, Operand X and Operand Y	User Sync select	Processor Clock select	Display Page select
RUN	**Fast**	**Pause**	**Break**	**Run**	**Opcode**	**Oper X**	**Oper Y**
	On/Off Toggle 10× Faster Clock and Sync	Program Execution Pause / Resume	Terminate Program Execution	RUN Program From Program Memory	—	—	—
ALT					User Sync select	Processor Clock select	Display Page select
PGM	**AddrSet**	**- Addr**	**Addr +**	**Dep +**	**Opcode**	**Oper X**	**Oper Y**
	Address set from Opcode, Operand X, Operand Y	Decrement Program Memory Address	Increment Program Memory Address	Write 12-bit word to Program Memory and inc PC	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)
ALT	Delete the current word (move all subsequent words down)	Reset Page and Pgm Memory Address to 0x000	Preset Program Mem Address to the last word used	Duplicate the current word (move all subsequent words up)			
	ALT+Both Keys → Pressed	Clear All Memory					

Gray = **ALT** key pressed

按钮命令

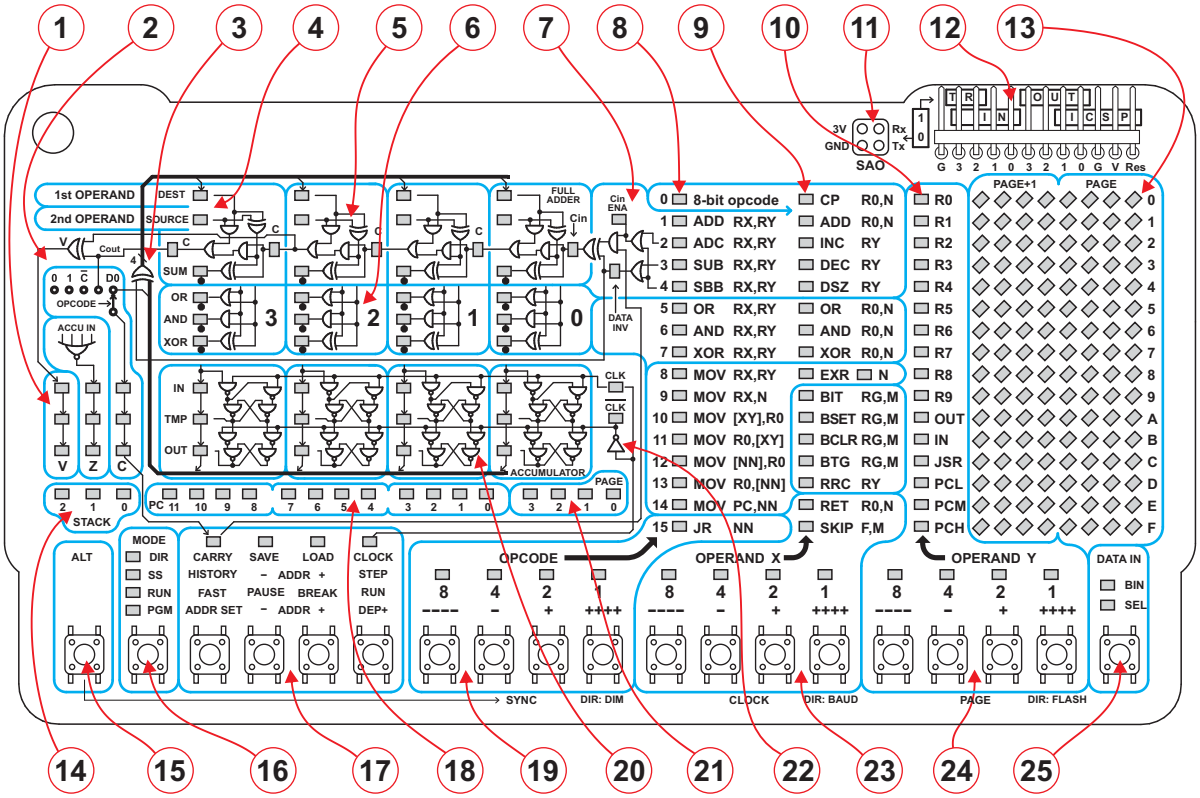
MODE	命令键				操作码	操作系统X	操作Y
DIR	携带	Save	Load	时钟	操作码	第十号歌剧X	歌剧Y
	切换携带Flag	Send 程序存储器以串行端口	Load 程序存储器 from 串行端口	硕士时钟源	指令操作码 (bits（第11-8段)	直接操作数 X or 操作码 (bits（第7段至第4段)	直接操作数Y (bits 3-0)
ALT		Save 程序存储器以选择 Flash	Load 程序存储器 from 选择 Flash		调光水平选择	Baud Rate 选择	Flash 部分 select for 保存/Load
SS	历史	- Addr	Addr +	Step	操作码	第十号歌剧X	歌剧Y
	进入历史子模式	减量程序存储器地址	增量程序存储器地址	执行 one 指令	指令操作码 (bits（第11-8段)	直接操作数 X or 操作码 (bits（第7段至第4段)	直接操作数Y (bits 3-0)
ALT	切换进位标志	复位程序存储器地址到0x000	预设程序存储器地址到最后一词用	从操作码设置地 操作数X and 操作数Y	User Sync 选择	处理器时钟选择	显示 Page 选择
RUN	Fast	暂停	打破	Run	操作码	第十号歌剧X	歌剧Y
	开/关切换10×快时钟和 Sync	程序执行/恢复	终止程序执行	RUN 程序 From 程序存储器	—	—	—
ALT					User Sync 选择	处理器时钟选择	显示 Page 选择
PGM	地址集	- Addr	Addr +	Dep +	操作码	第十号歌剧X	歌剧Y
	地址集从操作码，操作数X，操作数Y	减量程序存储器地址	增量程序存储器地址	写入12位词来程序存储器 PC Inc	指令操作码 (bits（第11-8段)	直接操作数 X or 操作码 (bits（第7段至第4段)	直接操作数Y (bits 3-0)
ALT	删除当前词 (move所有后续（下）	重置页面和Pgm 存储器地址到0x000	预设程序 Mem Address 到最后一词用	复制当前词 (move所有后续（上）			

ALT+按下两个键 → 清除所有内存

灰色=按下ALT键

Indicators, buttons and connectors

1. **STATUS** register, with **V** (**oVerflow**), **Z** (**Zero**) and **C** (**Carry**) flags
2. **Flag Logic**, which generates signals for flags
3. **Data Buffer/Inverter**, switched by Carry Logic **(7)** and used for Adder/Subtractor
4. **Operands**, represented and indicated as inputs to **ALU** unit
5. **Arithmetic** unit (4-bit **Full Adder / Subtractor**) as a part of **ALU** unit
6. **Logic** unit (4-bit **OR / AND / XOR** gates) as a part of **ALU** unit
7. **Carry Input Logic** (used for **Data** and **Carry Inversion** in case of subtraction)
8. **Opcode Decoder** output (also used as interactive **Code Disassembler**)
9. **Operand X Decoder** output (also used as interactive **Code Disassembler**)
10. **Operand Y Decoder** output (also used as interactive **Code Disassembler**)
11. **SAO** (**"Shitty Add-On"**) connector, with **Ground**, **+3V** and **UART Rx/Tx** pins
12. **I/O** connector, with **Input** and **Output** ports and **PIC MCU** Programming pins
13. **LED Matrix** which displays two pages (**2×16** nibbles) of **Data Memory**



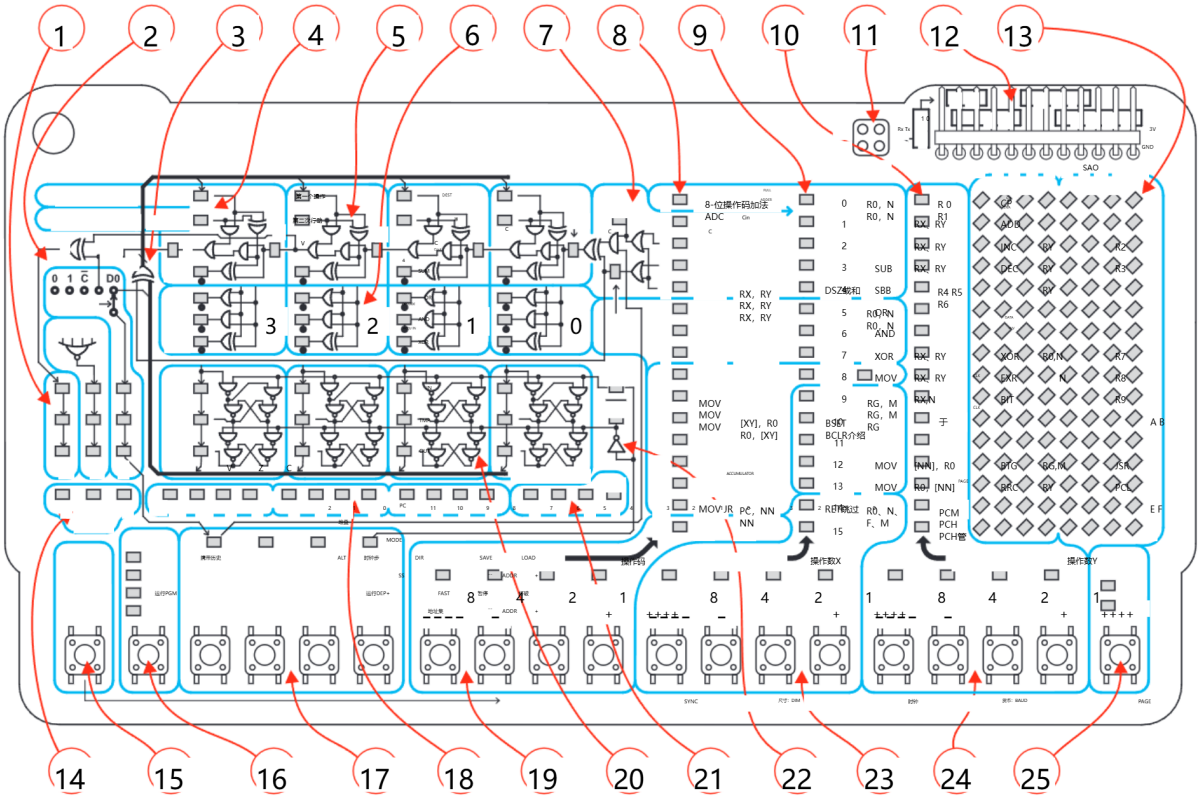
14. Three-bit **Stack Pointer** indicator
15. **ALT** button, which switches some indicators and buttons to alternate functions
16. **MODE** button, used to switch between **Direct/Single Step/Run/Program** mode
17. **Command Group** of buttons, with Mode-specific fuctions
18. **Program Memory Address** pointer (a.k.a. **Program Counter**, **PC**)
19. **Opcode** buttons and indicators (instruction bits **11-8**)
20. Symbolic representation and indicators for **Accumulator** register
21. **Page** register, determines which page of **Data Memory** is shown on LED matrix
22. **Master Clock** signal inverter, used for **Master-Slave Flip-Flops** triggering
23. **Operand X** buttons and indicators or **Opcode** extension (instruction bits **7-4**)
24. **Operand Y** buttons and indicators (instruction bits **3-0**)
25. **Data In Select** button and indicator, switches between **Binary** and **Select** mode

指示器、按钮和连接器

1. 状态寄存器，带有V（oVerflow）、Z（Zero）和C（Carry）标志
2. 标志逻辑，为标志生成信号
3. 数据缓冲器/反相器，由进位逻辑（7）切换，用于加法器/减法器
4. 操作数，表示和指示为ALU单元的输入

算术单元（4位全加器/减法器）作为ALU单元6的一部分。逻辑单元（4位OR / AND / XOR门）作为ALU单元7的一部分。进位输入逻辑（用于减法情况下的数据和进位反转）8. 操作码解码器输出（也用作交互式代码反汇编器）

操作数X解码器输出（也用作交互式代码反汇编器）10.操作数Y解码器输出（也用作交互式代码反汇编器）11. SAO（“Shitty Add-On”）连接器，带接地、+3V和12个Rx/Tx引脚。I/O连接器，输入和输出端口和PIC MCU编程引脚13. LED矩阵显示两页（2×16半字节）数据存储器



14.三位堆栈指针指示器15. ALT按钮，用于将某些指示器和按钮切换到备用功能

16. 模式按钮，用于在直接/单步/运行/程序模式之间切换
17. 命令按钮组，具有模式特定功能
18.程序存储器地址指针（也称为程序计数器，PC）
19.操作码按钮和指示灯（指令位11-8）
20.累加器寄存器的符号表示和指示符
21.页面寄存器，确定LED矩阵上显示的数据存储器页面
22.主时钟信号反相器，用于主从触发器触发
23.操作数X按钮和指示器或操作码扩展（指令位7-4）
24.操作数Y按钮和指示器（指令位3-0）
25. 数据输入选择按钮和指示灯，在二进制和选择模式之间切换

Indicators, buttons and connectors

On-Off Button

The only button available at the bottom side is the **On-Off** button. Switching **On-Off** is possible in every mode, and also when the user's program is running. In the **Off** state, clock signal is halted, the system processor is in Sleep mode and all outputs on the **I/O** connector (12) are in the high-impedance state. However, there are **pull-up** resistors on all inputs and **pull-down** resistors on all outputs. The only exception is the serial **Tx** output, which does not have not **pull-down**, but **pull-up** resistor, as the default level of **Tx** is high. The resistance of every pull-up and pull-down is **22KΩ**.

Switching the unit **Off** does not affect processor's registers or contents of memory or program state, so when the unit is turned **On** again, it will continue the execution as if it wasn't stopped at all.

ALT Button (15)

This is the only button that does not initiate the command execution when it is pressed, but it modifies the functions of other buttons and some indicators. It should be used similarly to the **Alt** button on the computer's keyboard, which means that it should be pressed prior to the button which function should be modified. The main Alt-functions of other buttons is printed under the button bar, and it is depended on the mode selected. Here is the function of buttons in the **Opcode**, **Operand X** and **Operand Y** fields, with the original and modified function in different modes:

	OPCODE buttons and indicator bar		**OPERAND X** buttons and indicator bar		**OPERAND Y** buttons and indicator bar	
MODE	Original	ALT pressed	Original	ALT pressed	Original	ALT pressed
DIR	Opcode	Dimmer	Operand X	Baud rate	Operand Y	Flash Addr
SS	Opcode	Sync	Operand X	Clock	Operand Y	Page
RUN	—	Sync	—	Clock	—	Page
PGM	Opcode	Sync	Operand X	Clock	Operand Y	Page

DATA IN Button and Indicators (25)

Used for **Data Input Method** selection, which affects button groups **Opcode**, **Operand X** and **Operand Y**. The same Data Input Method selection is valid for ALT-functions of the three groups (**Sync, Clock, Page, Dimmer, Baud Rate** and **Flash Address**). Every press of the **Data In** button toggles between the two methods, and the current method is displayed on the **BIN/SEL** indicators.

Here is the description of the two available methods:

1. BINARY Method

In the **Binary** method, every buttons simply toggles the corresponding bit state of the 4-bit selection register, displayed by four indicator above the buttons (Most Significant Bit, marked by "8", is at the left). At the same time, the 16-step indicator bar displays the decoded binary state of the selection register.

2. SELECT Method

Buttons **"-"** and **"+"** are used to decrement and increment by one the 4-bit binary state of the 4-bit selection register. Buttons **"----"** and **"++++"** can be used to decrement and increment the state of the same register by four, which can be used to speed up the selection.

指示器、按钮和连接器

开关按钮

底部唯一可用的按钮是开-关按钮。开关是可能的，在每一种模式，也当用户的程序正在运行。在关闭状态下，时钟信号停止，系统处理器处于睡眠模式，I/O连接器（12）上的所有输出均处于高阻抗状态。但是，所有输入端都有上拉电阻，所有输出端都有下拉电阻。唯一的例外是串行Tx输出，它没有下拉电阻，但上拉电阻，因为Tx的默认电平为高。每个上拉和下拉电阻为22 K Ω。

关闭单元不会影响处理器的寄存器或内存内容或程序状态，因此当单元再次打开时，它将继续执行，就好像它根本没有停止一样。

ALT按钮（15）

这是唯一一个在按下时不启动命令执行的按钮，但它会修改其他按钮和某些指示器的功能。它应该类似于计算机键盘上的Alt按钮，这意味着它应该在修改功能的按钮之前按下。其他按钮的主要Alt功能打印在按钮栏下，取决于所选的模式。以下是操作码、操作数X和操作数Y字段中按钮的功能，在不同模式下具有原始和修改后的功能：

	OPCODE按钮和指示条		OPERAND X按钮和指示条		OPERAND Y按钮和指示条	
MODE	原始	ALT按下	原始	ALT按下	原始	ALT按下
DIR	操作码	调光	操作数X	波特率	操作数Y	闪存地址
SS	操作码	Sync	操作数X	时钟	操作数Y	Page
RUN	—	Sync	—	时钟	—	Page
PGM	操作码	Sync	操作数X	时钟	操作数Y	Page

数据输入按钮和指示灯（25）

用于数据输入法选择，它影响按钮组“操作码”、“操作数X”和“操作数Y”。相同的数据输入方法选择对三个组（同步、时钟、页面、调光器、波特率和闪存地址）的ALT功能有效。每次按下数据输入按钮都在两种方法之间切换，当前方法显示在BIN/SEL指示器上。

以下是两种可用方法的描述：

1.二值化方法

在二进制方法中，每个按钮简单地切换4位选择寄存器的相应位状态，通过按钮上方的四个指示器显示（最高有效位，标记为“8”，位于左侧）。同时，16级指示条显示选择寄存器的解码二进制状态。

2. SELECT方法

“-”和“+”用于将4位选择寄存器的4位二进制状态递减和递增1。“-”和“+”可用于将同一寄存器的状态递减和递增4，可用于加快选择速度。

Indicators, buttons and connectors

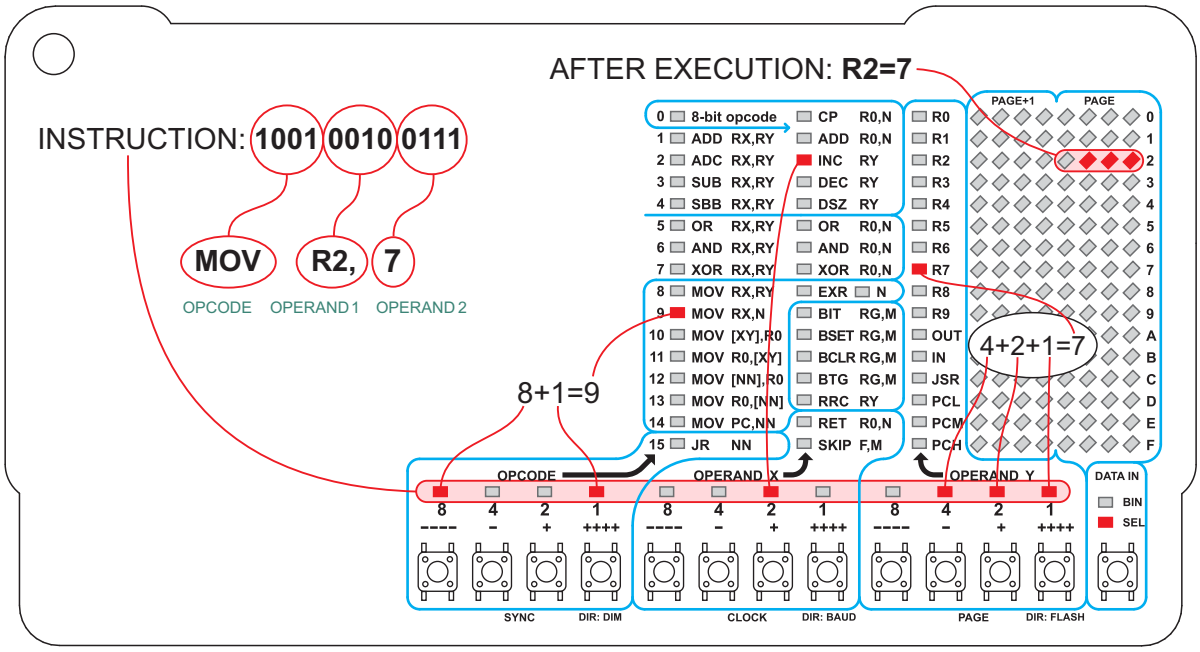
Preface: Bit coding

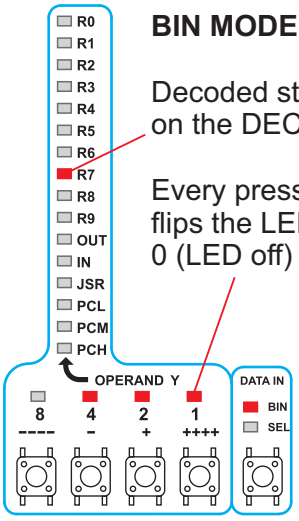
Before we start with the **Opcode** and **Operand** fields, just a few words about the bit coding fields inside the instruction word. As the unit is supposed to be programmed directly in a **machine language** (**binary code**, ones and zeros), special care was taken to simplify the structure of the instruction codes and pack the bit fields in 4-bit groups which are easy to perceive and remember.

The main code fields are **Opcode** and **Operands**. The typical instruction contains one opcode and a flexible number of operands.

Opcode (abbreviated from **op**eration **code**) is the portion of a machine language instruction that specifies the operation to be performed. In this case, the opcode is always located at the most significant bits of the 12-bit instruction word: bits 11-8 or bits 11-4.

Operands are values assigned to registers or memory and they are specified and accessed using more or less complex addressing modes. Here we have instructions which contain **one** or **two** operands (but note that there are processors which support more or zero operands). Generally, **Operands** contain **data**, and **Opcode** tells the processor **what to do** with the data.

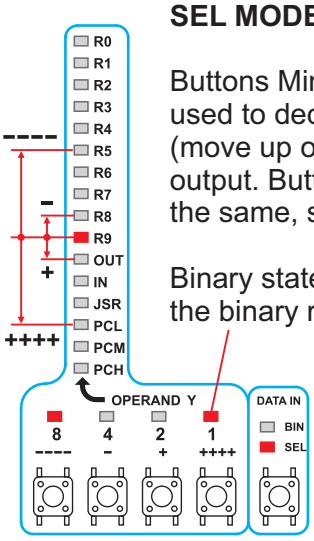




BIN MODE:

Decoded state is displayed on the DECODER column

Every press on the button flips the LED state:
0 (LED off) or 1 (LED on)



SEL MODE:

Buttons Minus (-) and Plus (+) are used to decrement or increment (move up or down) the decoded output. Buttons (----) and (++++) do the same, skipping 4 values at once

Binary state is displayed on the binary row

指示器、按钮和连接器

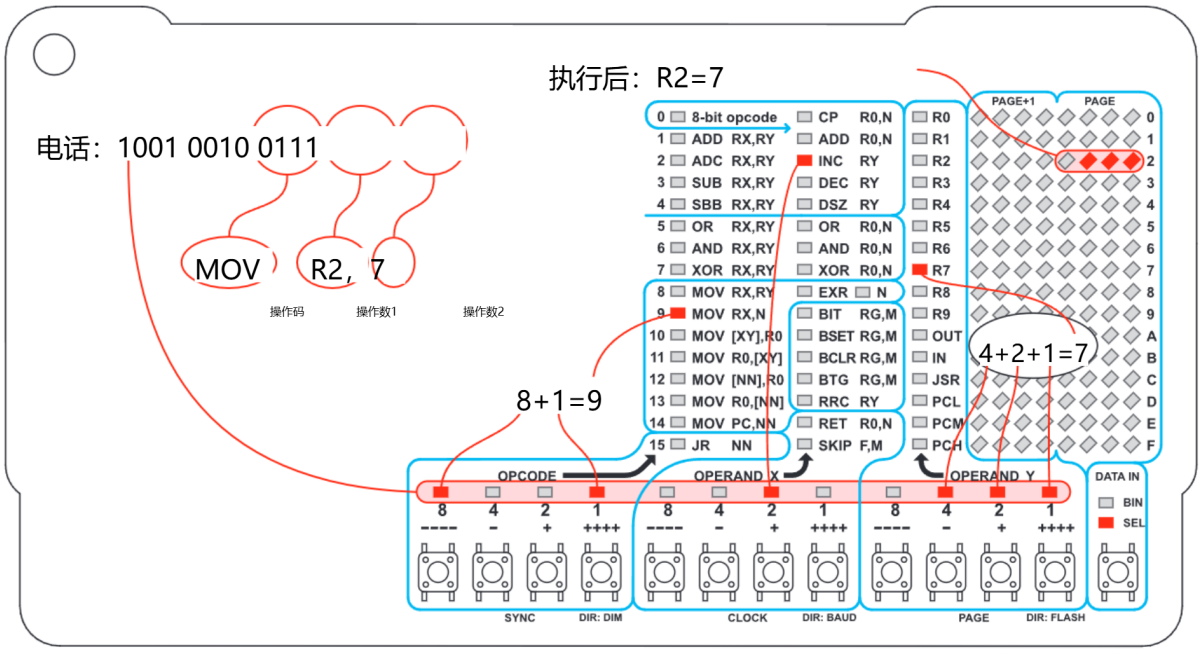
前言：位编码

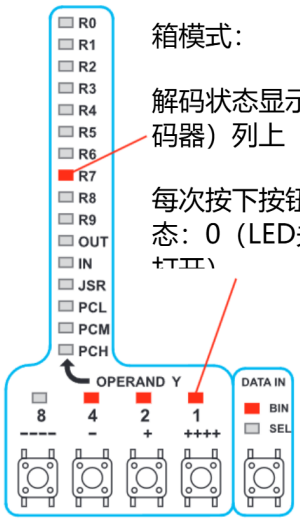
在我们开始介绍操作码和操作数字段之前，先简单介绍一下指令字中的位编码字段。由于该单元应该直接用机器语言（二进制代码，1和0）编程，因此特别注意简化指令代码的结构，并将位字段打包为易于感知和记忆的4位组。

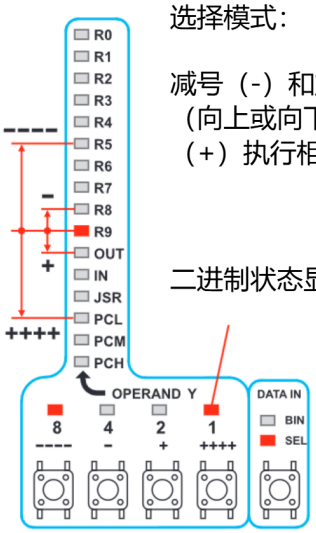
主要的代码字段是操作码和操作数。典型的指令包含一个操作码和一个灵活的操作数。

操作码（Opcode，缩写为operation code）是机器语言指令中指定要执行的操作的部分。在这种情况下，操作码总是位于12位指令字的最高有效位：位11-8或位11-4。

操作数是分配给寄存器或内存的值，它们是使用或多或少复杂的寻址模式指定和访问的。这里我们有包含一个或两个操作数的指令（但请注意，有些处理器支持更多或零个操作数）。通常，操作数包含数据，操作码告诉处理器如何处理数据。

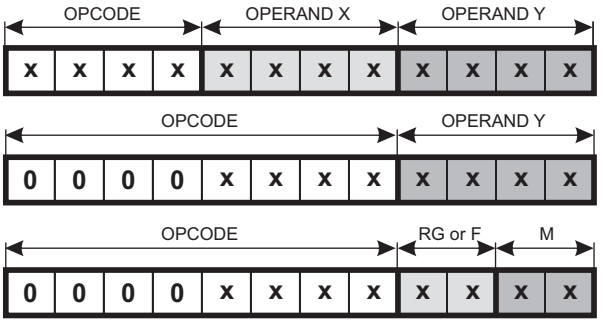






Indicators, buttons and connectors

Let's see how the instruction is organized internally. In this processor, every instruction has the same length, which is **12 bits**. Opcode and operands may be located in any part of the instruction code, but in our case, for clarity and ease of programming, the opcode always contains four or eight bits and it is located in the leftmost part of the 12-bit instruction word (highest order bits), and operand or operands are in the rightmost eight or four bits:



x = bit 0 or 1

If the opcode is eight bits wide, then the first four bits are always **0000**. This way of coding gives enough space for a total of **15** instructions with **4-bit** opcode (**0001**-**1111**), and **16** instructions with **8-bit** opcode (**00000000**-**00001111**).

Please note that there are several instructions (**Bit Test/Set/Clear/Toggle** and **Skip**) with the **Operand Y** field split in two 2-bit operands.

OPCODE Buttons and Indicators (19) and decoder (8)

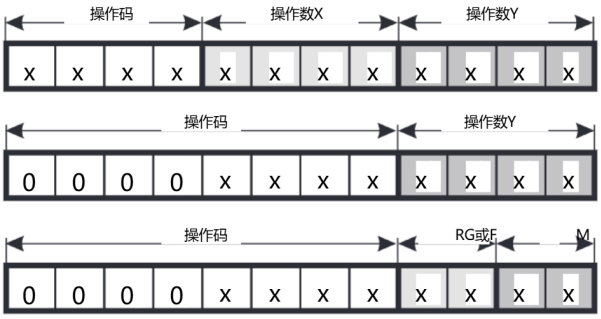
Buttons in the field which is conditionally named **Opcode** are generally used to preset the four upper (most significant) bits of the 12-bit instruction word. The word in binary form is readable on the indicators in the **Opcode (19)** field. The indicator column **(8)** represents the decoded 4-bit nibble from the Opcode field, and it has only one LED in ON state. This column can be conveniently used as the disassembled opcode with the instruction printed next to the LED.

If the **DATA IN** indicator is in the **BIN** state, Opcode can be entered bit-by-bit, and if it is in the **SEL** state, the same buttons are used to move the decoded output up/down by one (buttons **"-"** and **"+"**) or four places (buttons **"----"** and **"++++"**).

In **Single Step** and **RUN** modes, every modification of the **Program Memory Address (PC)** will automatically read the contents of the instruction and update the **Opcode, Operand X** and **Operand Y** indicators. You can modify it using buttons, end even execute the new state in **SS** mode (by pressing button **STEP**), but it will not affect the contents of the Program Memory. The only way to alter the contents of the Program Memory is to press the **DEPosit** button in **PGM** mode.

指示器、按钮和连接器

让我们看看指令是如何在内部组织的。在这个处理器中，每条指令都有相同的长度，即12位。操作码和操作数可以位于指令代码的任何部分，但在我们的情况下，为了清晰和易于编程，操作码总是包含四位或八位，并且它位于12位指令字的最左边部分（最高位），操作数位于最右边的八位或四位：



x =位0或1

如果操作码是8位宽，那么前四位总是0000。这种编码方式为总共15条具有4位操作码（0001-1111）的指令和16条具有8位操作码（0000000 -00001111）的指令提供了足够的空间。

请注意，有几个指令（位测试/设置/清除/切换和跳过）的操作数Y字段分为两个2位操作数。

OPCODE显示器和指示器（19）和解码器（8）

在有条件地命名为操作码的字段中的位通常用于预置12位指令字的四个高（最高有效）位。二进制形式的字在操作码（19）字段的指示器上可读。指示器列（8）表示来自操作码字段的解码的4位半字节，并且它只有一个LED处于ON状态。此列可以方便地用作反汇编操作码，其指令打印在LED旁边。

如果DATA IN指示器处于BIN状态，则可以逐位输入操作码，如果处于SEL状态，则使用相同的按钮将解码输出向上/向下移动一位（按钮“-”和“+”）或四位（按钮“-”和“+”）。

在单步执行和运行模式下，程序存储器地址（PC）的每次修改都将自动读取指令内容并更新操作码、操作数X和操作数Y指示器。您可以使用按钮修改它，结束甚至在SS模式下执行新状态（通过按下按钮STEP），但它不会影响程序存储器的内容。更改程序存储器内容的唯一方法是在PGM模式下按下存款按钮。

Indicators, buttons and connectors

OPERAND X Buttons and Indicators (23) and decoder (9)

Buttons and indicators in the **OPERAND X** group are used to preset the central nibble (bits 7-4) of the instruction. If the **Opcode** field contains bits 0000, **Operand X** field does not represent the operand anymore, but the extension of the **Opcode** field.

The word in binary form is readable on the indicators in the **Operand X (23)** field. The indicator column **(9)** represents the decoded 4-bit nibble from the **Operand X** field, and it has only one LED in ON state. This column can be conveniently used as the disassembled opcode, but the special care should be taken if the **Opcode** field contains **0000**, as the instruction printed next to the LED is valid only in that case. In all other cases, the decoded output should be treated as the register name printed next to the LED matrix field (**Data Memory**).

If the **DATA IN** indicator is at the **BIN** state, **Operand X** can be entered bit-by-bit, and if it is in the **SEL** state, the same buttons are used to move the decoded output up/down by one (buttons **"-"** and **"+"**) or four places (buttons **"----"** and **"++++"**).

In **Single Step** and **PGM** modes, every modification of the **Program Memory Address (PC)** will automatically read the contents of the instruction and update the **Opcode, Operand X** and **Operand Y** indicators. You can modify it using buttons, end even execute the new state in **SS** mode (by pressing button **STEP**), but it will not affect the contents of the Program Memory unless you press the **DEPosit** button in **PGM** mode.

Column **(9)** contains one extra indicator, which is in the instruction **EXR R0,N** field. This instruction exchanges a group of **General Purpose** and **Special Purpose** registers from the **Page 0** with the equivalent number of nibbles in the **Page 14** of Data Memory, and the indicator is flipped every time when the instruction is executed. So the indicator should be **ON** only when register contents are exchanged between **Page 0** and **Page 14** and **OFF** when they are flipped back to their original positions. That couls help keeping track of program execution.

In **Single Step** and **RUN** modes, the function of **Operand X** buttons and indicators is modified when the **ALT** button is depressed. In that case, **Clock** register is accessible instead of **Operand X** register. **Clock** register is used to adjust the processor speed.

OPERAND Y Buttons and Indicators (24) and decoder (10)

Buttons and indicators in the **OPERAND Y** group are used to preset the lower nibble (bits 3-0) of the instruction. The word in binary form is readable on the indicators in the **Operand Y (24)** field. The indicator column **(10)** represents the decoded 4-bit nibble from the **Operand Y** field, and it has only one LED in ON state. This column can be conveniently used as the disassembled opcode, and it is valid for most instructions, but not for instructions **Bit Test/Set/Clear/Toggle** and **Skip**, which split the field which we named as **Operand Y** in two 2-bit operands. For these instructions, decoding should be performed manually.

If the **DATA IN** indicator is at the **BIN** state, **Operand Y** can be entered bit-by-bit, and if it is in the **SEL** state, the same buttons are used to move the decoded output up/down by one (buttons **"-"** and **"+"**) or four places (buttons **"----"** and **"++++"**).

In **Single Step** and **PGM** modes, every modification of the **Program Memory Address (PC)** will automatically cause reading of the contents of the new instruction and update the **Opcode, Operand X** and **Operand Y** indicators. You can modify it using buttons, end even execute the new state in **SS** mode (by pressing button **STEP**), but it will not affect the contents of the Program Memory unless you press the **DEPosit** button in **PGM** mode.

In **Single Step** and **RUN** modes, the function of **Operand Y** buttons and indicators is modified when the **ALT** button is depressed. In that case, **Page** register is accessible instead of **Operand Y** register.

指示器、按钮和连接器

OPERAND X显示器和指示器（23）和解码器（9）

OPERAND X组中的指针和指示符用于预置指令的中央半字节（位74）。如果操作码字段包含位0000，则操作数X字段不再表示操作数，而是操作码字段的扩展。

二进制形式的单词在操作数X（23）字段的指示器上可读。指示器列（9）表示来自操作数X字段的解码的4位半字节，并且它只有一个LED处于ON状态。该列可以方便地用作反汇编操作码，但如果操作码字段包含0000，则应特别注意，因为LED旁边打印的说明仅在这种情况下有效。在所有其他情况下，解码输出应视为LED矩阵字段（数据存储器）旁边打印的寄存器名称。

如果DATA IN指示器处于BIN状态，则可以逐位输入操作数X，如果处于SEL状态，则使用相同的按钮将解码输出向上/向下移动一位（按钮“-”和“+”）或四位（按钮“-”和“+”）。

在单步和PGM模式下，每次修改程序存储器地址（PC）将自动读取指令内容并更新操作码、操作数X和操作数Y指示器。您可以使用按钮修改它，结束甚至在SS模式下执行新状态（通过按下按钮STEP），但它不会影响程序存储器的内容，除非您在PGM模式下按下存款按钮。

列（9）包含一个额外的指示符，其在指令EXR R0，N字段中。该指令将第0页中的一组通用和专用寄存器与数据存储器第14页中相同数量的半字节交换，并且每次执行该指令时，该指示符都会翻转。因此，只有当寄存器内容在第0页和第14页之间交换时，指示器才应亮起，而当寄存器内容翻转回原始位置时，指示器才应熄灭。这有助于跟踪程序执行情况。

在单步和运行模式下，当按下ALT按钮时，操作数X按钮和指示器的功能被修改。在这种情况下，可以访问时钟寄存器而不是操作数X寄存器。时钟寄存器用于调节处理器速度。

运算符Y和指示器（24）和解码器（10）

OPERAND Y组中的指针和指示符用于预置指令的低位半字节（位30）。二进制形式的字在操作数Y（24）字段的指示器上可读。指示器列（10）表示来自操作数Y字段的解码的4位半字节，并且它只有一个LED处于ON状态。此列可以方便地用作反汇编操作码，并且它对大多数指令有效，但不适用于位测试/设置/清除/切换和跳过指令，这些指令将我们命名为操作数Y的字段拆分为两个2位操作数。对于这些指令，应手动执行解码。

如果DATA IN指示器处于BIN状态，则可以逐位输入操作数Y，如果处于SEL状态，则使用相同的按钮将解码输出向上/向下移动一位（按钮“-”和“+”）或四位（按钮“-”和“+”）。

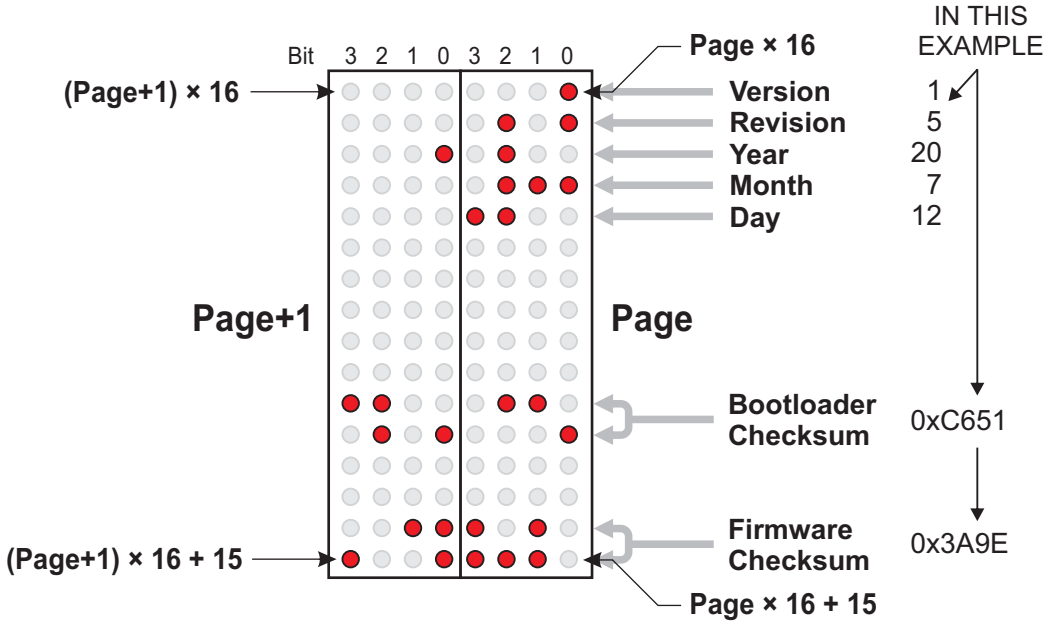
在单步和PGM模式下，每次修改程序存储器地址（PC）将自动导致阅读新指令的内容，并更新操作码、操作数X和操作数Y指示器。您可以使用按钮修改它，结束甚至在SS模式下执行新状态（通过按下按钮STEP），但它不会影响程序存储器的内容，除非您在PGM模式下按下存款按钮。

在单步和运行模式下，当按下ALT按钮时，操作对象Y按钮和指示器的功能被修改。在这种情况下，可以访问页寄存器而不是操作数Y寄存器。

Indicators, buttons and connectors

LED Matrix (Data Memory) (13) and Page indicator (21)

Data Memory display is visually organized as **16×8** matrix, but it is functionally divided in two **16×4** displays. The right **16×4** half displays the contents of one Data Memory page defined by the state of the **Page** register (21), and the left half is for the next one (**Page+1**). The whole Data Memory contains **256 nibbles**, which gives a total of **16** pages, and if the right half displays the last page (**Page 15**), then the left half is wrapped to the beginning of the address space as **Page 0**. This enables watching both **General Function Registers** (on **Page 0**) and **Special Function Registers** (**Page 15**) at the same time.



Data Memory display (13) is disabled in **DIRect** and **PGM** modes, but in **DIRect** mode it has the special function when the **ALT** button is pressed. Then it displays the occupancy of **16 Flash Memory blocks**, which can help in **Flash Memory** organization and navigation.

Also, after the **Master Reset,** Data Memory Display in **DIRect** mode (which is default after **Reset**) shows the **Version/Revision/Year/Month/Day** numbers or the firmware release at the first five rows of the **LED** Matrix. In the middle of the matrix (rows **10** and **11**) there is the **Checksum** of Program Memory for the **Bootloader Segment**, and, on the two bottom rows, the **Checksum** for the **General Segment** (main firmware).

Master Reset is possible only after the batteries are disconnected and then reconnected, or when pins **G** (Ground) and **Res** (Reset) of the **I/O Connector** are shortened externally. After any button is pressed, this data is cleared from the display.

Note that **Master Reset** also clears all **Program** and **Data Memory**, but not the contents saved in the internal Flash.

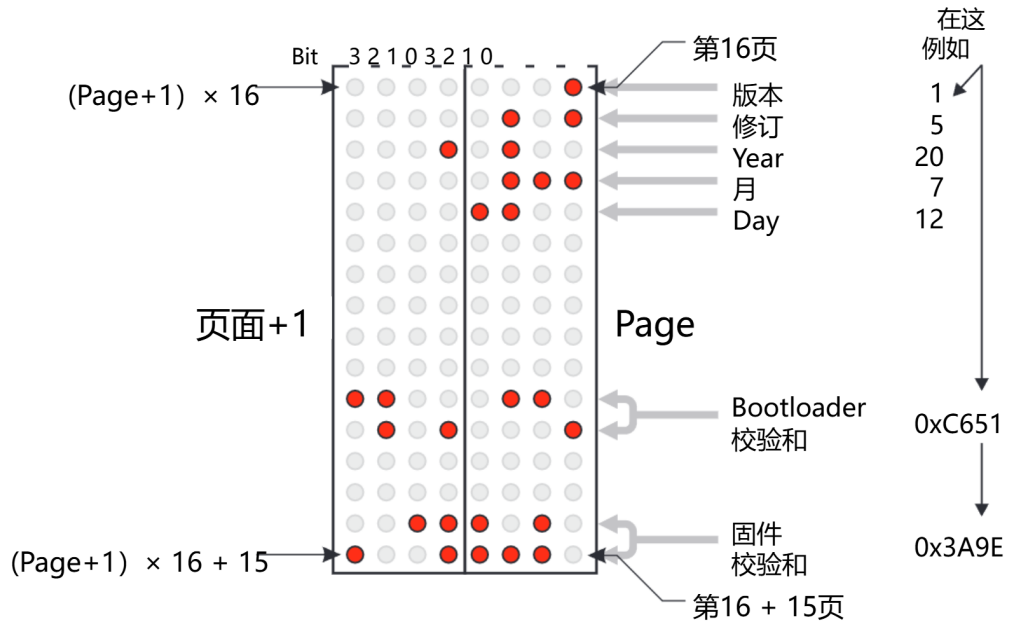
Data Memory display can be disabled under the program control, if bit **2** (**MatrixOff**) in the register **WrFlags** (Address **0xF3**) is set. If bit 3 (**LedsOff**) in the register **WrFlags** (Address **0xF3**) is set, all other **LEDs** will be disabled, only the **LED CLK** or **INVERSE CLK** (on the schematic drawing) will still be **ON**. These **LEDs** are the indicator that the unit is in operation (or, if they are alternatively blinking, that it is running), and they can not be disabled.

Page is the 4-bit register which is accessible to the user's program in the **Special Function Register** (**SFR**) group, in the data memory address **0xF0**. In modes **SS** and **RUN** it can be easily preset manually, when the **ALT** button is depressed and the **Operand Y** buttons are used for Page contents adjusting.

At every program **Run**, the **Page** register is reset to **0000**. Mode **Single Step** (**SS**) has its own **Page** register, so if some value was preset in **SS** mode, it will be kept and restored at every reentry to the **SS** mode.

LED矩阵（数据存储器）（13）和页面指示灯（21）

数据存储器显示器在视觉上组织为16×8矩阵，但在功能上分为两个16×4显示器。16 x 4的右半部分显示由页面寄存器（21）的状态定义的一个数据存储器页面的内容，左半部分用于下一个页面（Page+1）。整个数据存储器包含256个半字节，总共有16页，如果右半部分显示最后一页（第15页），则左半部分将被绕到地址空间的开头，作为第0页。这样可以同时监视通用功能寄存器（第0页）和特殊功能寄存器（第15页）。



在DIRect和PGM模式下，数据存储器显示（13）被禁用，但在DIRect模式下，按下ALT按钮时，它具有特殊功能。然后显示16个闪存块的占用情况，这有助于闪存的组织和导航。

此外，在主机复位后，DIRect模式下的数据存储器显示（复位后的默认设置）在LED矩阵的前五行显示版本/修订/年/月/日编号或固件版本。在矩阵的中间（第10和11行），有引导加载程序段的程序存储器校验和，在底部的两行，有通用段（主固件）的校验和。

只有在断开电池并重新连接之后，或者在I/O连接器的引脚G（接地）和Res（复位）从外部缩短时，才可以进行主复位。按下任何按钮后，此数据将从显示器上清除。

请注意，主复位也清除所有程序和数据存储器，但不保存在内部闪存的内容。

如果设置了寄存器“数据存储器标志”（地址0xF 3）中的位2（矩阵关闭），则可以在程序控制下禁用数据存储器显示。如果设置寄存器“闪烁标志”（地址0xF 3）中的位3（LED关闭），则所有其他LED将被禁用，只有LED CLK或INVERSE CLK（在原理图上）仍将亮起。这些LED指示装置正在运行（或者，如果它们交替闪烁，则表示装置正在运行），并且它们不能被禁用。

页是4位寄存器，可供特殊功能寄存器（SFR）组中的用户程序访问，位于数据存储器地址0xF 0中。在SS和RUN模式下，当按下ALT按钮并使用操作数Y按钮进行页面内容调整时，可以轻松手动预设。

每次程序运行时，页面寄存器重置为0000。模式单步（SS）有自己的页面寄存器，因此如果在SS模式中预设了某个值，则该值将在每次重新进入SS模式时保留并恢复。

Indicators, buttons and connectors

1st and 2nd OPERAND (Input to ALU unit) (4)

This is the first field of the **ALU/Accumulator** data flow, which is in the core of the processor. The data indicated in this field actually do not exist as registers, but only displays the input states to the ALU, and thus makes it easier to follow the process.

Please note the difference between these pairs of terms:
"**Operand X**" - "**Operand Y**",
"**1st Operand**" - "**2nd Operand**", and
"**Source**" - "**Destination**"

In many cases these pairs of terms will mean the same, but there are also cases when there is the difference. Operands **X** and **Y** are simply operands which are defined in **X** and **Y** fields on the panel, and that's all. At the other hand, **1st** and **2nd** operands are just defined by the order of appearance, and **Source** and **Destination** are exactly what these words mean.

There are different rules for different processors, but here the most popular rule is applied. If there are two operands, then the first one is always the destination (sometimes it's also the source), and the second one is always the source. So if the instruction is:

ADD RX, RY

that means "Add Arithmetically contents of **Register Y** to the contents of **Register X** and write the result in **Register X**".

Some instructions have only one operand. In general case, it is source and destination at the same time. For instance:

INC RY

means "Increment the value of **Register Y** by one and write the result in **Register Y**". It's obvious that there was an invisible source, which is the literal "1", added to the **Register Y**.

Sometimes the destination is hidden as the operand, and you have to know the operation defined by the Opcode, to know where the result is stored:

BIT R2, 3

This operation tests bit **3** in register **R2**, so it's the single bit source, but where is the destination? In this instruction, it is the single bit destination, **Flag Z**. To make it more complicated, if bit **3** in register **R2** is **0**, the resulting flag **Z** will be **1**, and vice versa. But it makes more sense when we know that **Z (Zero) Flag** is set **(1)** when the result of the operation is **Zero (0)**.

It is clear that it is not easy to define the operands precisely, so the representation of input signal to the **ALU** will sometimes be inaccurate. Also, captions **"DEST"** and **"SOURCE"** next to the indicators are only roughly informative.

4-bit Full Adder / Subtractor (5)

Arithmetic unit performs adding and subtracting operations either with **unsigned positive**, or **Two's complement signed** (either positive or negative) binary numbers. **"Full"** Adder means that it not only adds bits, but also processes **Carry (C)** bit. So every bit stage has three inputs (**A**, **B** and **Carry** from the previous stage) and two outputs (**Sum** and **Carry** to the next stage). The Carry input of the **LSB** (Least Significant Bit) is the **global Carry input** to the adder, and the output from the **MSB** (Most Significant Bit) is the **global Carry output**.

Subtracting is actually adding of negative value, so the source operand is **inverted**. To make it negative in **2's complement** form, it should be also incremented by one, but the inverse **Carry** logic (which is named **Borrow**) in subtraction process compensates this and always gives the correct result, even without adding. If there is the **Borrow** condition (**no Carry**), then the resulting **-1** difference (caused by non-adding **1** at negation) automatically adds **Borrow** (**-1**) to the result, and **No Borrow** (**Carry set**) adds **1** and again compensates **2's complement** negation.

Adder/Subtractor is used not only for **Add** and **Subtract** instructions (with or without **Carry** or **Borrow**), but also for **CP (Compare)** instruction. Compare is actually same as **Subtract**, but the result is not written anywhere but lost, only the flags are preserved.

第一和第二运算符（输入到ALU单元）（4）

这是ALU/累加器数据流的第一个字段，位于处理器的核心中。
此字段中指示的数据实际上不作为寄存器存在，而只是显示ALU的输入状态，因此更容易跟踪过程。

请注意这两对术语之间的区别：
“操作数X”-“操作数Y”、“第一操作数”-“第二操作数”和“源”-“目的地”

在许多情况下，这些术语对的意思是相同的，但也有情况下，有差异。操作数X和Y是在面板上的X和Y字段中定义的简单操作数，仅此而已。另一方面，第一个和第二个操作数只是由出现的顺序定义的，而Source和Destination正是这些词的意思。

对于不同的处理器有不同的规则，但这里应用的是最流行的规则。如果有两个操作数，那么第一个总是目标（有时也是源），第二个总是源。所以如果指令是：

ADD RX，RY

这意味着“将寄存器Y的算术内容与寄存器X的内容相加，并将结果写入寄存器X”。

有些指令只有一个操作数。在一般情况下，它同时是源和目的地。例如：

INC RY

表示“将寄存器Y的值递增1，并将结果写入寄存器Y”。很明显，有一个看不见的源，这是文字“1”，添加到寄存器Y。

有时目的地被隐藏为操作数，你必须知道操作码定义的操作，才能知道结果存储在哪里：

BIT R2，3

此操作测试寄存器R2中的位3，因此它是单个位源，但目的地在哪里？在此指令中，它是单位目的地，标志Z。为了使其更复杂，如果寄存器R2中的位3为0，则结果标志Z将为1，反之亦然。但是当我们知道当操作的结果为零（0）时，Z（Zero）Flag被设置（1）时，这更有意义。

显然，精确定义操作数并不容易，因此ALU的输入信号表示有时会不准确。此外，指标旁边的标题“目的地”和“来源”只是粗略的信息。

4-10位全加器/减法器（5）

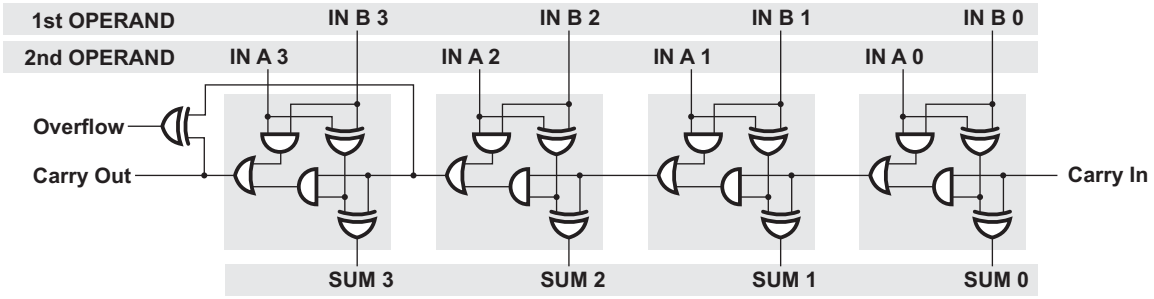
算术单元执行加法和减法操作，无论是与无符号的正，或二进制补码有符号（无论是正或负）的二进制数。
“全”加法器意味着它不仅增加位，而且还处理进位（C）位。因此，每个位级都有三个输入（A、B和来自前一级的进位）和两个输出（求和和进位到下一级）。LSB（最低有效位）的进位输入是加法器的全局进位输入，MSB（最高有效位）的输出是全局进位输出。

减法实际上是负值的加法，因此源操作数被反转。为了使它在2的补码形式中为负，它也应该增加1，但是减法过程中的反向进位逻辑（称为Borrow）补偿了这一点，并且总是给出正确的结果，即使没有添加。如果存在Borrow条件（无进位），则产生的-1差（由求反时不加1引起）自动将Borrow（-1）添加到结果中，而No Borrow（进位设置）添加1并再次补偿2的补码求反。

加法器/减法器不仅用于加和减指令（带或不带进位或借位），还用于CP（比较）指令。Compare实际上与Subtract相同，但结果不会写入任何地方，而是会丢失，只有标志会保留。

Indicators, buttons and connectors

It's amazing to learn about the theory of operation of the binary adder/subtractor. Two's complement binary math sometimes looks like magic, when everything turns simple with inverting and negating binary numbers and processing them always in the same adder/subtractor hardware. The Carry logic not only "works" for adding and subtracting, but also for **signed numbers** processing in the same hardware.



In a few words, signed numbers in the **2's complement** math are represented so that **MSB** (leftmost bit) is the **sign** (**"0"** for **"+", "1"** for **"-"**), and all other bits follow the 2'complement rule (inverted bits plus **1**). So there is one bit less for the number representation, as the range for the 4-bit signed number is **-8** to **+7** (for the **8-bit** number, it would be **-128** to **+127**), but everything else is quite simple: adding and subtracting can be performed in the same adder/subtractor hardware!

Adding and subtracting of longer (unsigned positive or signed negative or positive) numbers is performed in serial manner, using **Carry/Borrow** bit for linking. This is the example how to add or subtract two **16-bit** numbers, wether they are unsigned positive or signed negative or positive. If one number is in registers **R0, R1, R2** and **R3**, and the another one in **R4, R5, R6** and **R7**, after this addition (or subtraction) the result will be in **R0, R1, R2** and **R3**:

Addition:	**ADD R0, R4**	Subtraction:	**SUB R0, R4**
	ADC R1, R5		**SBB R1, R5**
	ADC R2, R6		**SBB R2, R6**
	ADC R3, R7		**SBB R3, R7**

Note that the first operation is without **Carry** (or **Borrow**), and all others are with **Carry** (or **Borrow**). This method can be used to add or subtract numbers of any length.

The example uses the **Little Endian** notation (**Least Significant** nibbles are written in the lower address of memory or register file). The principle is the same for **Big Endian** notation (**Most Significant** nibbles on low addresses, **Least Significant** on top), but the order of registers would be reversed (**ADD R3, R7**, then **ADC R2, R6**, and so on). Lowest bits are always processed first.

The same technique is applicable for adding or subtracting of **signed numbers** of any length. The rule is that there is only one Sign bit for the number of any length, always at the **MSB** location of the Most Significant nibble.

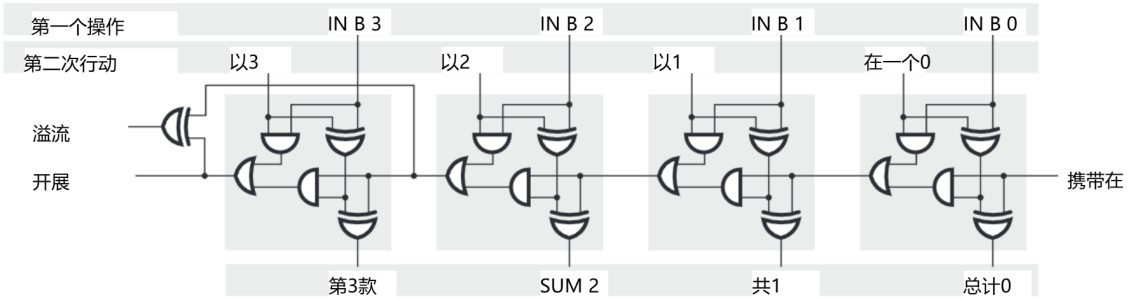
For the **unsigned binary numbers**, the global **Carry** (or **Borrow**) bit for the result is available after the last **ADC** (or **SBB**) instruction. If the **Carry Flag** is **set**, that means that the **addition has overflowed** and the result can't fit the register width. For **subtraction**, the outcome is **reversed**: **No Carry** (which means **Borrow Set**) means that the result **has overflowed** (the correct word here is **Underflowed**) and thus not usable.

This was valid for **unsigned** numbers, but for **signed** numbers **Carry Flag** outcome has no meaning, but the **V** (**Overflow**) **Flag** is used instead. So if **V Flag** is **set**, the result has overflowed or underflowed (can not be represented in the existing register) and it is not usable.

V Flag has no meaning for the **unsigned numbers**, so it is not frequently used. That's why it is not present as a condition in the **SKIP** instruction, but it is available and can be tested (using the instruction **BIT RG, M**) in the **SFR** (**Special Function Register**) named **RdFlags**, bit **1**. The Special Function Register **RdFlags** is at the **Data Memory** address **0xF3**.

Overflow Flag (**V**) is generated in the adder/subtractor hardware circuit in a very simple way: if the **Carry Input** to the last Adder bit and the **Carry Output** from the same adder bit are different, the **Overflow Flag** is set, and that's all. So the single **XOR** circuit does the whole task, and it is just another example of the simplicity and beauty of the adder/subtractor circuit.

学习二进制加法器/减法器的运算原理是很令人惊奇的。二进制补码数学有时看起来像魔术，当一切都变得简单，反转和否定二进制数，并始终在相同的加法器/加法器硬件处理它们。进位逻辑不仅适用于加法和减法，而且适用于同一硬件中的有符号数处理。



简而言之，2的补码数学中的有符号数被表示为MSB（最左边的位）是符号（“0”表示“+"，“1”表示“-”），所有其他位都遵循2 '补码规则（反转位加1）。因此，数字表示法少了一位，因为4位有符号数字的范围是-8到+7（对于8位数字，它将是-128到+127），但其他一切都很简单：加法和减法可以在同一个加法器/减法器硬件中执行！

较长数字（无符号正数或有符号负数或正数）的加减以串行方式执行，使用进位/借位进行链接。这是如何添加或减去两个16位数的例子，无论他们是无符号的积极或有符号的消极或积极的。如果一个数在寄存器R 0、R1、R2和R3中，另一个在R4、R5、R6和R7中，在此加（或减）之后，结果将在R 0、R1、R2和R3中：

添加：	添加R0、R4	减法：	R0、R4
	ADC R1、R5		SBB R1、R5
	ADC R2、R6		SBB R2、R6
	ADC R3、R7		SBB R3、R7

请注意，第一个操作没有Carry（或Borrow），所有其他操作都有Carry（或Borrow）。此方法可用于添加或减去任何长度的数字。

该示例使用小端表示法（最低有效半字节写入内存或寄存器文件的较低地址）。大端表示法的原理相同（最高有效半字节位于低位地址，最低有效半字节位于高位地址），但寄存器的顺序相反（ADD R3、R7，然后是ADC R2、R6，依此类推）。最低位总是首先处理。

同样的技术也适用于任何长度的有符号数的加法或减法。
规则是任何长度的数字只有一个符号位，总是在最高有效半字节的MSB位置。
对于无符号二进制数，结果的全局进位（或借位）位在最后一个ADC（或SBB）指令之后可用。如果设置了进位标志，则意味着加法溢出，结果不能适合寄存器宽度。对于减法，结果是相反的：No Carry（意味着Borrow Set）意味着结果已经溢出（这里正确的词是Underflowed），因此不可用。

这对无符号数字有效，但对于有符号数字，Carry Flag结果没有意义，但使用V（溢出）标志。因此，如果设置了V标志，则结果已溢出或下溢（无法在现有寄存器中表示），并且它不可用。

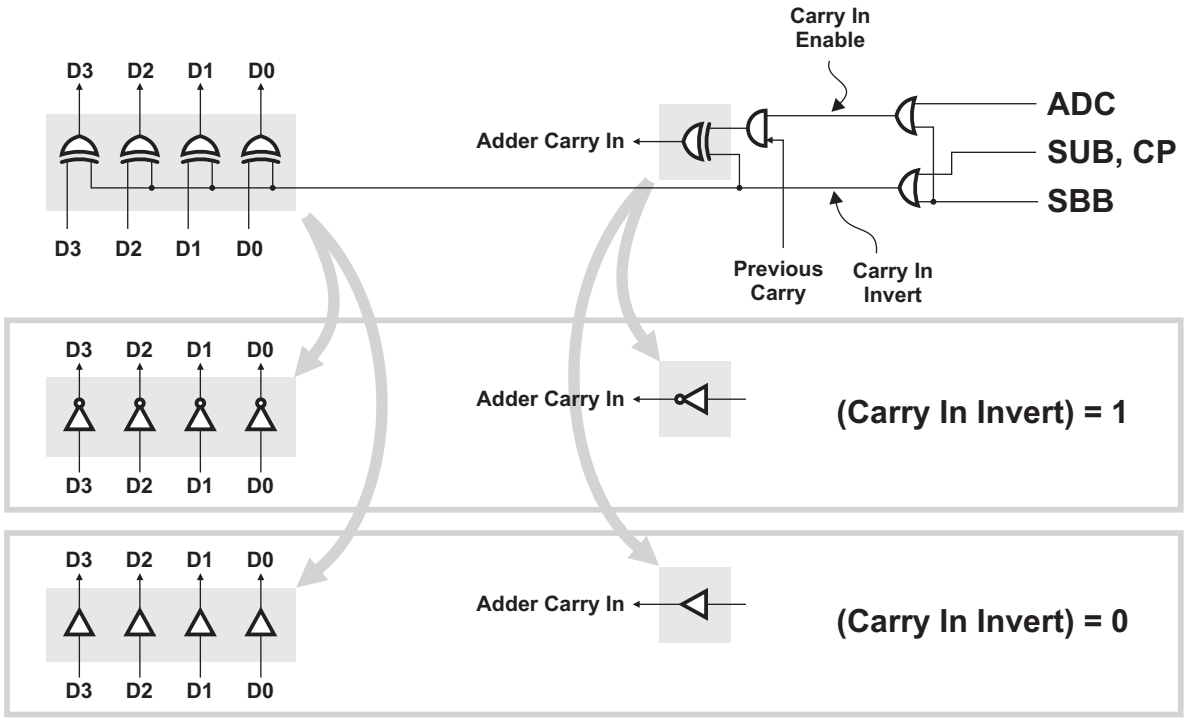
V Flag对于无符号数字没有意义，因此不经常使用。这就是为什么它不作为SKIP指令中的条件出现，但它是可用的，并且可以在名为RdFlags的SFR（特殊功能寄存器）位1中进行测试（使用指令BIT RG，M）。特殊功能寄存器RdFlags位于数据存储器地址0xF 3。

溢出标志（V）在加法器/加法器硬件电路中以一种非常简单的方式产生：如果最后一个加法器位的进位输入和来自同一个加法器位的进位输出不同，则设置溢出标志，仅此而已。因此，单个XOR电路完成了整个任务，这只是加法器/减法器电路简单而美观的另一个例子。

Indicators, buttons and connectors

Carry Input Logic (7) and Data Buffer/Inverter (3)

This is the simple logic circuit which inverts **Carry** logic level and **Data Bus signals** for subtraction, and leaves them unchanged (**true logic**) for addition. Also, it switches off the **Carry** input signal (forces it to **Low** for addition, or **High** for subtraction) in the operations which do not process Carry input signal (**ADD, SUB** and **CP**).



Logic **XOR** circuit, which is at the input of the **Adder** and which generates **Cin** signal, simply inverts the **Carry** signal if the **Carry In Invert =1**, and serves as a single buffer (which does not modify the signal level) if **Carry In Invert =0**. The same is valid for all **DATA** signals in the internal **DATA BUS** (there is only one representation of the Data Bus **XOR** circuit on the panel schematics). So both **Carry** and **Data** are inverted if the instruction involves **Subtraction**.

Logic unit (4-bit OR / AND / XOR gates) (6)

This is the second part of the **ALU** unit, where logic instructions are performed. Its structure is quite straightforward, but several facts should be noted before we finish the description of **ALU**.

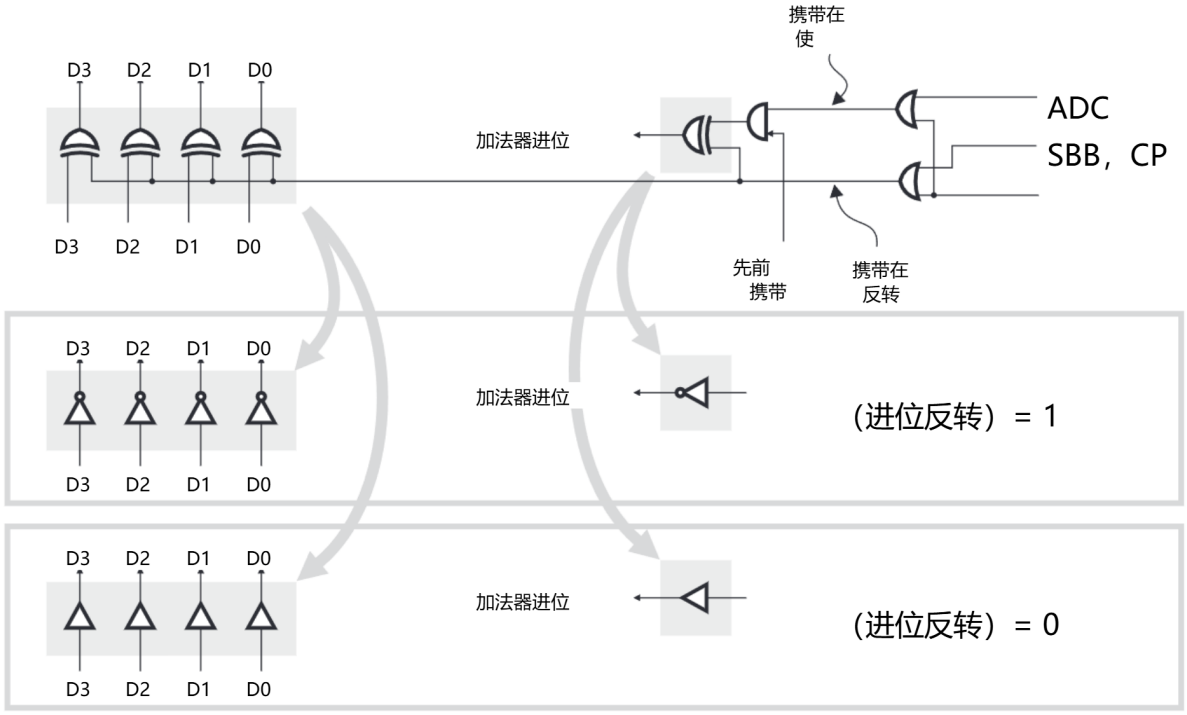
Due to the lack of space on the panel, the schematic of **ALU** circuit is simplified. One of the circuits that is missing is the complex part of the instruction decoder, which selects not only the result from the adder or some of logic outputs (**OR, AND** or **XOR**), but also the data path from **Data Memory** to the Accumulator inputs, the **I/O** data path, **SFR** logic and so on. The vast majority of the circuits are not represented, simply because it would, even in the simplest possible project, require the panel to be the size of the average table surface, with thousands of gates and many indicators.

Also, the **ALU** circuit is not optimized, but drawn so that it is clear and straightforward. The real **ALU** circuit in the microprocessor looks much less familiar and hard to follow and understand at the first sight, as the number of gates is minimized.

One more thing which is different in modern processors is the **Carry Generator** logic. This “serial” approach works correctly, but it slows down the operation of the processor, due to the propagation delays on the long path, through many gates from the **Carry Input** to the **Carry Output**. Note that there is also (but not represented here) the **Fast Carry Generator**, which works in parallel mode and thus requires more gates, but has much lower propagation delay.

进位输入逻辑（7）和数据缓冲器/反相器（3）

这是一个简单的逻辑电路，它将进位逻辑电平和数据总线信号反相，用于减法，并使它们保持不变（真逻辑），用于加法。此外，在不处理进位输入信号的操作（ADD、CLK和CP）中，它会关闭进位输入信号（将其强制为低电平进行加法，或将其强制为高电平进行减法）。



逻辑XOR电路位于加法器的输入端，并产生Cin信号，如果进位反转=1，则简单地反转进位信号，如果进位反转=0，则用作单个缓冲器（不修改信号电平）。这同样适用于内部数据总线中的所有数据信号（面板原理图上只有一个数据总线XOR电路的表示）。

因此，如果指令涉及减法，则进位和数据都被反转。

逻辑单元（4位OR / AND / XOR门）（6）

这是ALU单元的第二部分，执行逻辑指令。它的结构非常简单，但在我们完成对ALU的描述之前，应该注意几个事实。

由于面板上的空间不足，ALU电路的原理图被简化。缺少的电路之一是指令解码器的复杂部分，它不仅选择加法器的结果或一些逻辑输出（或、与或异或），而且还有从数据存储器到累加器输入的数据路径、I/O数据路径、SFR逻辑等等。绝大多数电路都没有表示出来，仅仅是因为即使在最简单的项目中，要求面板的大小为平均桌面的大小，具有数千个门和许多指示器。

此外，ALU电路不是优化的，而是绘制的，因此它是清晰和直接的。由于门的数量被最小化，微处理器中的真实的ALU电路看起来不那么熟悉，乍一看很难理解和理解。

现代处理器的另一个不同之处是进位生成器逻辑。这种“串行”方法工作正常，但由于从进位输入到进位输出的长路径上的传播延迟，它会减慢处理器的操作。请注意，还有（但这里没有表示）快速进位发生器，它工作在并行模式，因此需要更多的门，但具有更低的传播延迟。

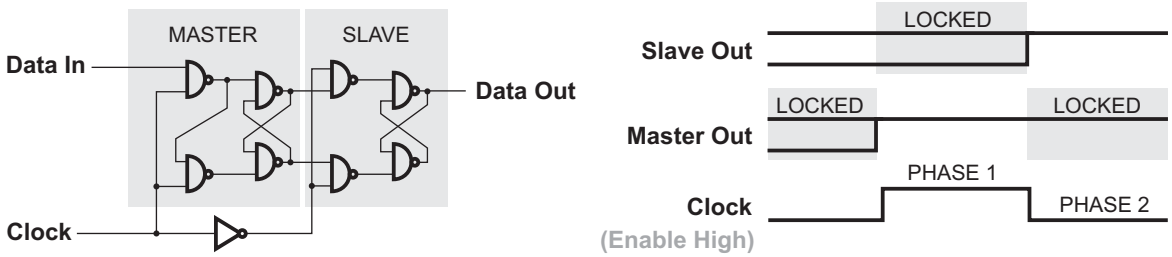
Indicators, buttons and connectors

Accumulator (20) and **Master Clock Signal Inverter** (22)

The first thing that should be noted is that there is not one, but **16 Accumulators** in this processor, and they are named as **Registers R0-R15**. So it's good to imagine them as **16 layers** of the accumulator schematics, and only one is selected and visible, depended which register is the destination one.

Please note that some instructions need no accumulator, as the destination may be the **Data Memory** nibble, **Program Counter**, or even a single bit (**Flag**) in the **Status Register**.

The accumulator contains a series of four Flip-Flops, not a simple ones, but **Master-Slave Edge-Triggered D Flip-Flops**.



In the first phase of the **Clock** signal on the **Master-Slave Edge-Triggered D Flip-Flop** schematics, when the **Clock** signal is **High** (in **Single Step** or **Direct** mode, it's when the button **Step** or **Clock** is pressed), the first **Flip-Flop** is in **transparent** state (unlocked). When the **Clock** logic level is changed to **Low**, the first Flip-Flop (the **Master** one) latches the **Data** logic level and the second (**Slave**) Flip-Flop is in **transparent** state. This two-fold latching solves the problem of circular self-triggering when the same register serves as the **source** and the **destination** at the same time, as one of Flip-Flops is always latched. The **Master Flip-Flop output** can change its state only when the **Clock** signal is **high,** but the **Slave** output can be changed only in the moment of the **falling edge** of the **Clock** signal. (Note: ignore **Enable** signal for now.)

Of course the best way would be to have the full schematics with **16 Accumulators**, but the available space on the panel allows only one. It must be switched with every new instruction, so that it displays the logic states of the register or memory location which is the current destination, and it may cause the unexpected switching of the output logic states of the Accumulator. For instance, when the button **Step** is first depressed in **Single Step** mode, the input logic states are transferred to the **Temporary Outputs** (outputs of **Master Flip-Flops**), but when the button is released, the same logic states would normally appear on **Accumulator Outputs**. However, the new instruction is read from the **Program Memory**, possibly with the new destination, and now the **Accumulator** represents the new register. That's why its logic states were unexpectedly changed.

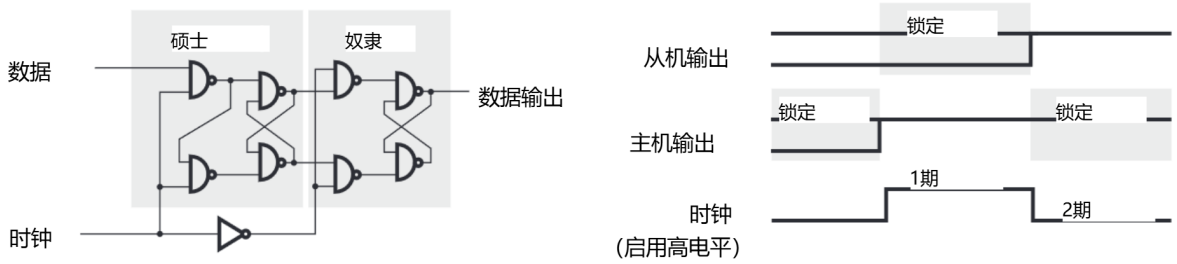
This does not happen in **Direct** mode, at least in most cases, when the destination stays the same, so it is much easier to follow the data flow in the **Direct** mode.

累加器（20）和主时钟信号反相器（22）

首先应该注意的是，在这个处理器中不是一个，而是16个累加器，它们被命名为寄存器R 0-R15。因此，最好将它们想象为累加器原理图的16层，并且只有一个被选择和可见，这取决于哪个寄存器是目标寄存器。

请注意，有些指令不需要累加器，因为目的地可能是数据存储器半字节，程序计数器，甚至是状态寄存器中的单个位（标志）。

累加器包含一系列四个触发器，不是简单的触发器，而是主从边缘触发D触发器。



在主从边沿触发D触发器的时钟信号的第一阶段，
如原理图所示，当时钟信号为高电平时（在单步或直接模式下，当按下步进或时钟按钮时），第一个触发器处于透明状态（解锁）。当时钟逻辑电平变为低电平时，第一个触发器（主触发器）锁存数据逻辑电平，第二个触发器（从触发器）处于透明状态。这种双重锁存解决了当同一寄存器同时用作源和目的地时的循环自触发问题，因为其中一个触发器总是被锁存。主触发器输出只能在时钟信号为高电平时改变其状态，而从触发器输出只能在时钟信号下降沿的时刻改变。(Note：暂时忽略启用信号。）

当然，最好的方法是有完整的示意图与16个蓄能器，但可用的空间在面板上只允许一个。它必须与每个新指令一起切换，以便它显示当前目的地的寄存器或存储器位置的逻辑状态，并且它可能导致累加器的输出逻辑状态的意外切换。例如，当在单步模式下第一次按下步进按钮时，输入逻辑状态被转移到临时输出（主触发器的输出），但当按钮被释放时，相同的逻辑状态通常会出现在累加器输出上。然而，新指令从程序存储器中读取，可能具有新的目的地，并且现在累加器表示新寄存器。这就是为什么它的逻辑状态会意外地改变。

在Direct模式下不会发生这种情况，至少在大多数情况下，当目标保持不变时，因此在Direct模式下更容易跟踪数据流。

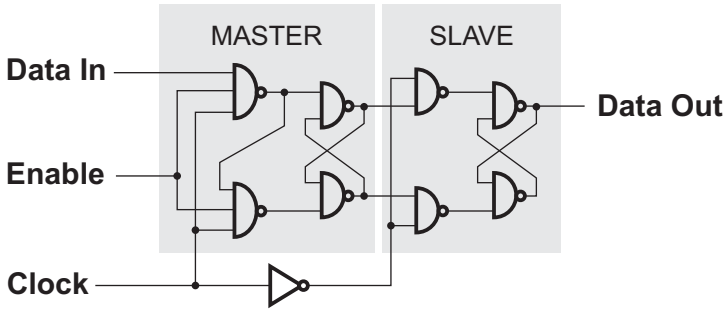
Indicators, buttons and connectors

STATUS register (1) and **FLAG logic** (2)

Status register, which contains **Overflow, Zero** and **Carry** flags, is probably the simplest, but among the most important parts of processor's hardware. Flags are kind of **decision-makers** in the program flow, and the **Carry Flag** is sometimes called the **1-bit Accumulator**.

Flags are propagated through the **Status Register** in a similar way as **Data Bits** are propagated through the **Accumulator**: there are three **Master-Slave Edge-Triggered D Flip-Flops**, which are triggered with the same **Clock** signal and at the same time as the **Accumulator Flip-Flops**. The **Status Flip-Flops** are not represented on the panel schematics in order to save space, but they are the same and triggered by the same **Clock** and **Inverse Clock** signals.

This is a good moment to say that there is the logic circuit which decides if the **Flip-Flop** will be clocked or not. Actually, the **Instruction Decoder** decides about that: some instructions have to keep the existing contents of the **Accumulator**, or individual **Flags**. Only when the **Accumulator** (to be more specific, the **addressed register**) is the destination of the operation, its contents should be clocked. If the current instruction does not affect **Flags** or **Accumulator** contents, the instruction decoder pulls the **Enable** signal low in the corresponding **Flip-Flop**(**s**), and thus locks the Flip-Flop state. The same is valid for every flag individually, as some instructions do not affect some flags, and they should be preserved safely in the **Flip-Flop**. The additional logic inputs, which disable the **Flip-Flop** clocking, is represented on the following schematics. There are the same **Latch Enable** inputs on the Accumulator Flip-Flops, as the contents of the Accumulator should be preserved in the case when it is not the destination (bit oriented instructions, program branching or compare instructions). Note that Enable inputs are not drawn on the simplified panel drawing.



We have seen how simply the **Overflow Flag** is generated: a single **XOR** logic circuit detects the single-bit logic equality at the **Carry Input** and **Carry Output** signals of the last stage of the **Adder**. **Zero Flag** logic is also simple, as it only tests the **Accumulator** input for **Zero**. There is only one exception: instruction **BIT RG, M** sets or resets the **Zero Flag** depended on the tested bit state. Note that **Zero** condition always sets **Zero Flag** to **Non-Zero**, and **Non-Zero** condition resets it to **Zero**. This is valid not only for **BIT RG, M** instruction, but also in every other case. If the result is **Zero** (all bits are **0000**), the **Zero Flag** will be set (**1**), and if one or more bits in the result are set (**1**), the Zero Flag will be reset (**0**). In a few words, **Zero Flag = 1** means **Zero**, and **Zero Flag = 0** means **Non-Zero**.

Carry Flag can be a result of arithmetical or bitwise rotation. It can also be unconditionally **set** by the instruction **OR R0,N**, **reset** (**AND R0,N**) or **toggled** (**XOR R0,N**). The drawing on the following page, which represents **Carry** signal flow, is valid for **ADD/ADC/SUB/SBB** instructions (including **CP** also, which is the same as **SUB,** only without storing the result), but a similar flow could be drawn for **RRC** (**Rotate Right Through Carry**) also.

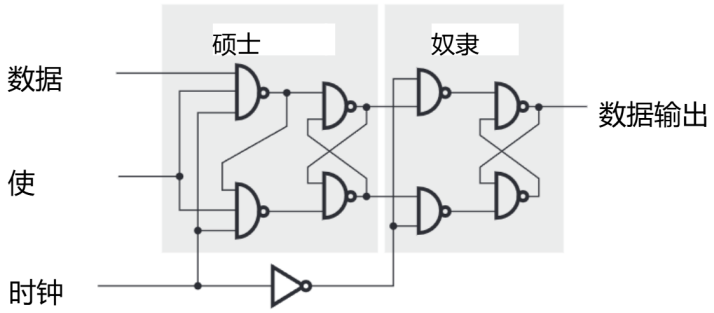
Carry Flag has a complex behavior, so it is represented in as much as nine indicators on the panel schematics. Please look the following page.

状态寄存器（1）和标志逻辑（2）

状态寄存器包含溢出、零和进位标志，它可能是处理器硬件中最简单的，但也是最重要的部分。标志是程序流中的决策者，进位标志有时被称为1位累加器。

标志通过状态寄存器传播的方式与数据位类似
通过累加器传播：有三个主从边缘触发D翻转-
触发器，由与累加器触发器相同的时钟信号同时触发。为了节省空间，状态触发器未在面板原理图上显示，但它们是相同的，并由相同的时钟和反时钟信号触发。

这是一个很好的时刻说，有逻辑电路，决定是否触发器将时钟或没有。实际上，指令解码器决定：一些指令必须保留累加器的现有内容，或单独的标志。只有当累加器（更具体地说，寻址寄存器）是操作的目的地时，其内容才应该被计时。如果当前指令不影响标志或累加器内容，则指令解码器将相应触发器中的使能信号拉低，从而锁定触发器状态。这同样适用于每个单独的标志，因为某些指令不影响某些标志，它们应该安全地保存在触发器中。禁用触发器时钟的附加逻辑输入如以下原理图所示。 累加器触发器上有相同的锁存使能输入，因为累加器的内容应在其不是目的地的情况下保留（面向位的指令、程序分支或比较指令）。请注意，启用输入未绘制在简化的面板图形上。

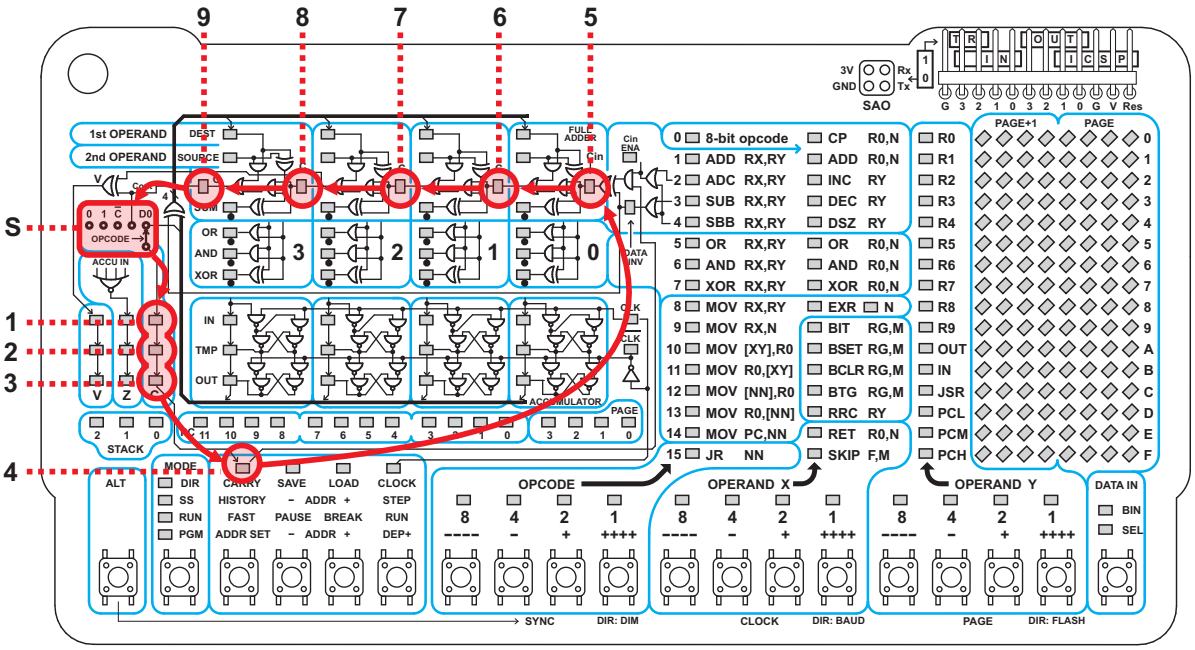


我们已经看到溢出标志的产生是多么简单：一个单一的XOR逻辑电路检测加法器最后一级的进位输入和进位输出信号的一位逻辑相等。零标志逻辑也很简单，因为它只测试累加器输入是否为零。只有一个例外：指令BIT RG，M根据测试位状态设置或重置零标志。请注意，零条件始终将零标志设置为非零，非零条件将其重置为零。这不仅适用于BIT RG，M指令，而且适用于所有其他情况。如果结果为零（所有位均为0000），则零标志将被设置（1），如果结果中的一个或多个位被设置（1），则零标志将被重置（0）。简单地说，Zero Flag = 1表示零，Zero Flag = 0表示非零。

进位标志可以是算术或按位旋转的结果。它也可以通过指令OR R 0，N、reset（AND R 0，N）或togged（XOR R 0，N）无条件设置。下一页的图表示进位信号流，对ADD/ADC/UART/SBB指令有效（也包括CP，与UART相同，只是不存储结果），但也可以为RRC（通过进位向右旋转）绘制类似的流。

进位标志具有复杂的行为，因此它在面板原理图上用多达9个指示器表示（注：现在忽略启用信号。请看下面的页面。

Indicators, buttons and connectors

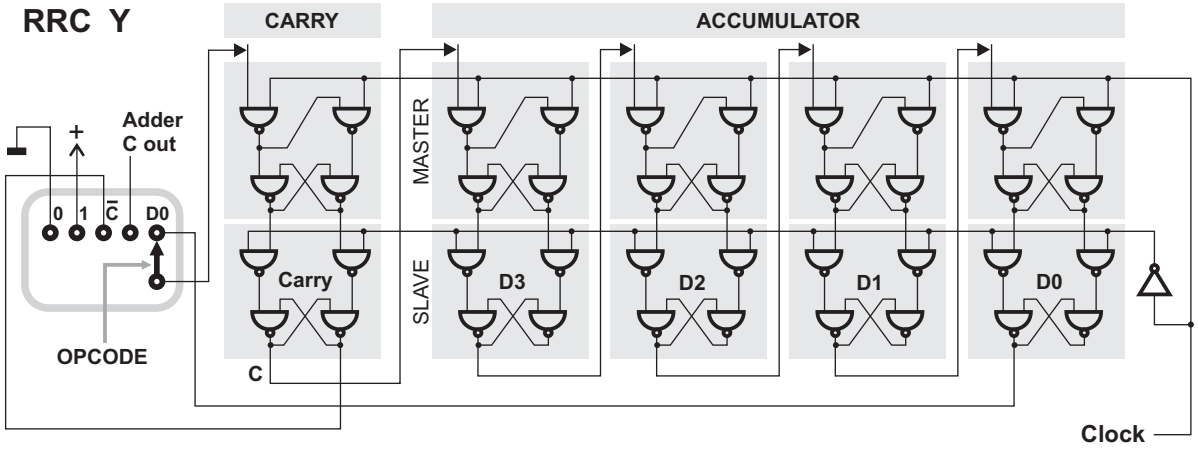


Nine indicators represent the same flag, and yet every one has the different meaning. After the selector **S**, which is under the control of the instruction decoder, the 1-bit content of the Carry Flag is fed to the input of **Carry Flip-Flop** (1) in the **Status** register. If the instruction is supposed to affect the **Carry Flag**, the **Clock** signal will first load the content to the **TMP** point (2) (**Master Flip-Flop** output), and then to the **Carry Output** (3) of the **Status** register.

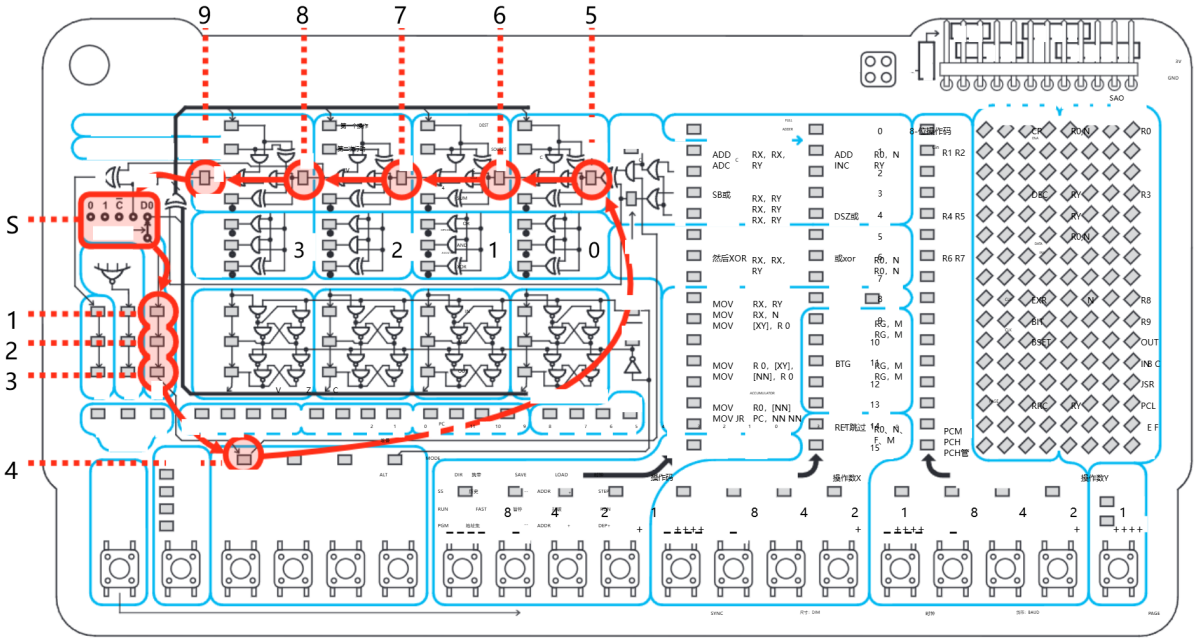
At the same moment, the **Carry Flag LED** (4), which is in the **Command Buttons** field, fetches the same state. This indicator is not a part of the typical processor, but it is included in this model as it allows the user to modify the **Carry Flag** state manually, for experimenting. So you can watch interactively how **Carry Flag** affects the **Adder** states.

After passing through the simple logic, which inverts **Carry Flag** in the case of subtraction and turns it off if no Carry input is needed, there are **Intermedial Carry Flags** (6), (7), (8) between the **Adder** stages, before the final **Carry** (9) is generated.

In the case of the **RRC Y** instruction, the data flow of the **Carry Flag** is not represented here in detail, but there is the **D0** (**Data 0**) signal extracted from the internal **Data Bus**, which is fed to the rightmost contact of the **Selector S**. So the **D0** logic state is driven to the **Carry Flag**, and the rest of the data flow is represented on the following schematics diagram:



指示器、按钮和连接器

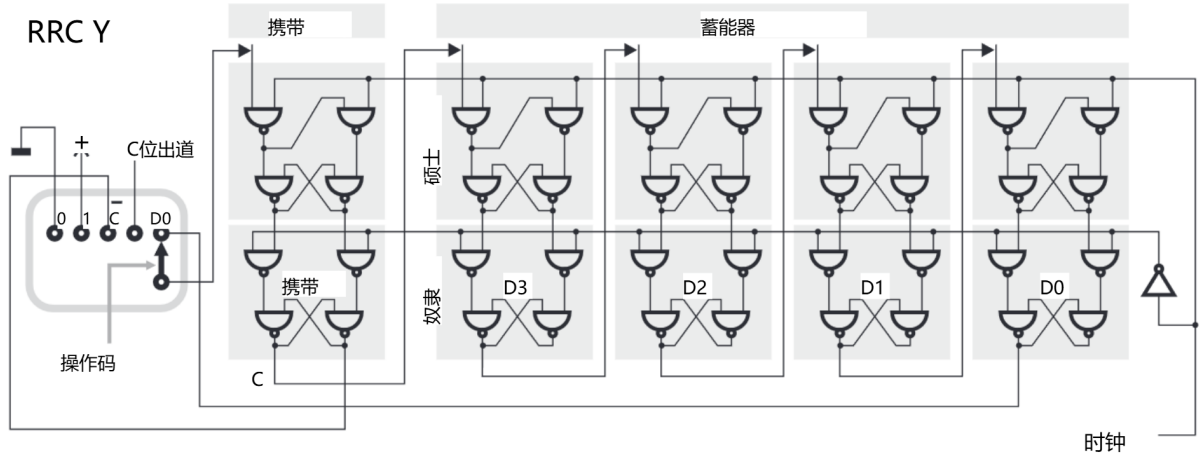


九个指标代表同一面旗帜，但每一个指标都有不同的含义。在选择器S之后，在指令解码器的控制下，进位标志的1位内容被馈送到状态寄存器中进位触发器（1）的输入。如果指令应该影响进位标志，时钟信号将首先将内容加载到TMP点（2）（主触发器输出），然后加载到状态寄存器的进位输出（3）。

与此同时，位于命令行字段中的进位标志LED（4）读取相同的状态。此指示器不是典型处理器的一部分，但它包含在此型号中，因为它允许用户手动修改进位标志状态，以便进行实验。因此，您可以交互式地观察进位标志如何影响加法器状态。

在通过简单逻辑之后，在产生最终进位（9）之前，在加法器级之间存在中间进位标志（6）、（7）、（8），该简单逻辑在减法的情况下反转进位标志，并且如果不需要进位输入则将其关闭。

在RRC Y指令的情况下，进位标志的数据流在这里没有详细表示，但是存在从内部数据总线提取的D0（数据0）信号，该信号被馈送到GPRS的最右侧触点。因此，D0逻辑状态被驱动为进位标志，其余数据流如以下原理图所示：



Stack Pointer (SP) (14)

This three-bit register is used to address the Data Memory where the **Program Counter** (**PC**) will be stored during the execution of the subroutine (writing to **JSR** General Purpose Register), and restored back to Program Counter at the execution of RETURN (**RET R0,N** instruction).

Program Counter contains **12** bits (**3** Data Memory locations), so one Stack position requires **3** nibbles for storing. When the subroutine is called (when the program writes a nibble in **JSR** General Purpose Register, on location **0x0C**), **PC** is stored at the Data Memory location **0x10+3×[SP]** (**low-order** address nibble first). Then the **SP** register is incremented by one, and **JSR**, **PCM** and **PCH** registers are copied to the **PC** (**JSR** is the **low-order** address nibble).

When the Return from subroutine (instruction **RET R0,N**) is executed, literal **N** is loaded to the register **R0**, then the **SP** is decremented by one, and contents of Data Memory from the three locations (the first one is 0x10+3×[SP]) is written back to the **PC**. Note that both **SP** and **PC** registers are not available directly to the user's program.

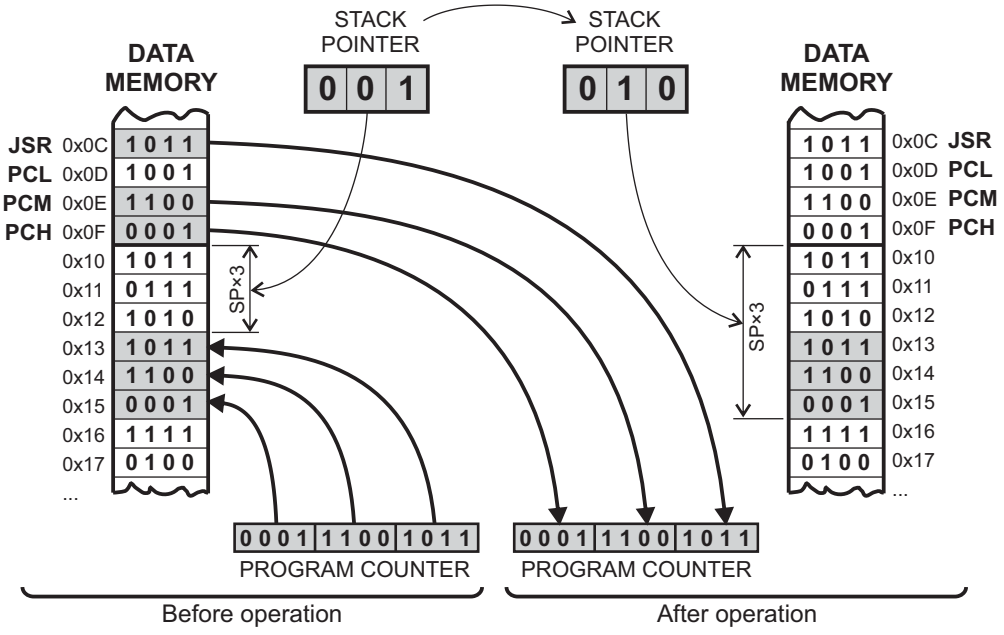
A total of **5** levels of Stack Pointer can be used in the program, which means that only the first **15** Data Memory locations from the **Page 1** will be used for the Stack storage. If the **SP** overflows to **110** (decimal **6**), which will happen if subroutines are called **6** times without executing **RET R0,N**, program execution will be halted and the **Error** condition will be indicated, so that the Stack indicator will blink at the value **110** (which is the attempted value when the error occurred). The Error condition should be cleared by pressing any key (the command assigned to the key will not be executed).

If more Returns (instructions **RET R0,N**) are executed than Calls (writing to the register **JSR**), register **SP** will be in the **underflow** condition. Program execution will be halted and the Error condition will be indicated, so that the Stack indicator will blink at the value **111** (decimal **-1** in signed notation). All register indicators and **Data Memory** matrix are still active, so the user can see the **PC Address** and other conditions under which the error occurred. The **Error** condition should be cleared by pressing any key (the command assigned to the key will not be executed).

When the unit is in the **RUN** mode, Stack Pointer is automatically cleared at every program **RUN** and program **Break**, and when the **Error** state is cleared. In **Single Step (SS)** mode, clearing **Data Ram** (performed when **Program Counter** is cleared by pressing **ALT-ADDR minus**), which also clears the **Stack**.

When entering **Single Step** (**SS**) mode, **Stack** restores the last value which it had in the **SS** mode.

Here is how **Stack Pointer** addresses **Data Memory** when the subroutine is called (when the value is written in the register **JSR**). The process is inverse when the instruction **RET R0, N** is executed.



堆栈指针（SP）（14）

此三位寄存器用于寻址数据存储器，在执行子例程（写入JSR通用寄存器）期间，程序计数器（PC）将存储在该数据存储器中，并在执行RETURN（RET R 0，N指令）时恢复回程序计数器。

程序计数器包含12位（3个数据存储器位置），因此一个堆栈位置需要3个半字节进行存储。当调用子例程时（当程序将半字节写入JSR通用寄存器中的位置0x0C时），PC存储在数据存储器位置0x10+3×[SP]（先写入地址半字节）。然后SP寄存器递增1，JSR、PCM和PCH寄存器被复制到PC（JSR是低位地址半字节）。

当执行从子例程返回（指令RET R0，N）时，将文字N加载到寄存器R0，然后SP减1，并将数据存储器的内容从三个位置（第一个是0x10+3×[SP]）写回PC。请注意，SP和PC寄存器都不能直接用于用户程序。

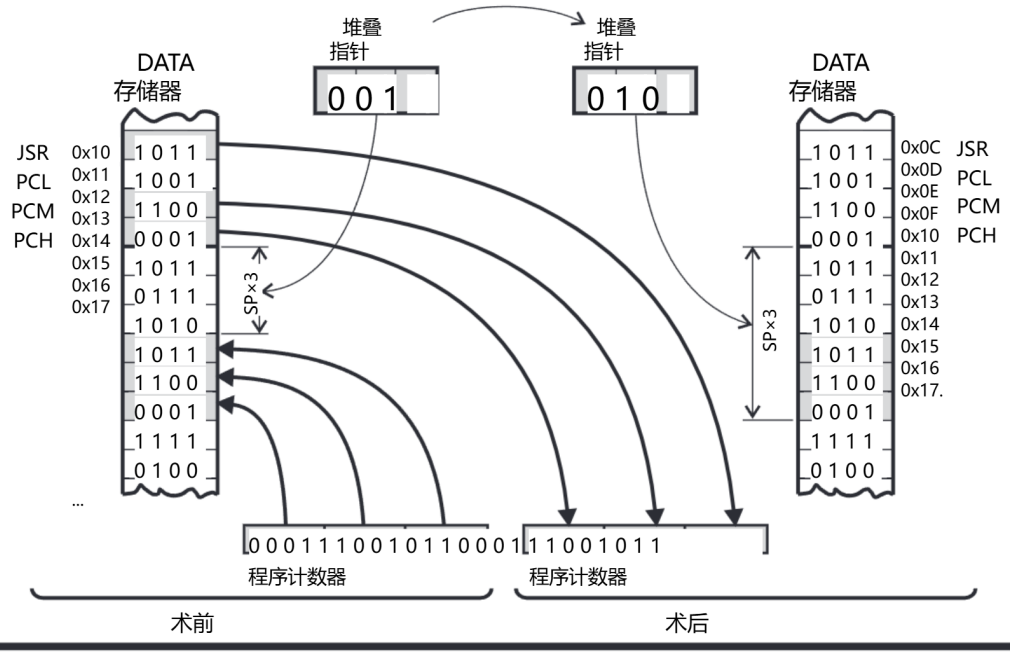
在程序中总共可以使用5个级别的堆栈指针，这意味着只有第一个
15 第1页的数据存储器位置将用于堆栈存储。如果SP溢出到
110（十进制6），如果子程序被调用6次而没有执行RET R0，N，则会发生这种情况，程序执行将停止，并指示错误条件，因此堆栈指示器将在值110（错误发生时的尝试值）处闪烁。按任意键应清除错误条件（分配给该键的命令将不会执行）。

如果执行的返回（指令RET R 0，N）比调用（写入寄存器JSR）多，则寄存器SP将处于下溢状态。程序执行将被停止，错误条件将被指示，因此堆栈指示器将在值111（十进制-1符号表示法）处闪烁。所有寄存器指示器和数据存储器矩阵仍然有效，因此用户可以看到PC地址和发生错误的其他条件。按任意键应清除错误条件（分配给该键的命令将不会执行）。

当装置处于运行模式时，堆栈指针在每个程序运行和程序中断时自动清除，并且当错误状态被清除时。在单步（SS）模式下，清除数据RAM（当按下ALT-ADDR减清除程序计数器时执行），这也会清除堆栈。

当进入单步（SS）模式时，堆栈恢复它在SS模式下的最后一个值。

下面是当调用子例程时（当值写入寄存器JSR时）堆栈指针如何寻址数据内存。当指令RET R0，N被执行时，该过程是相反的。



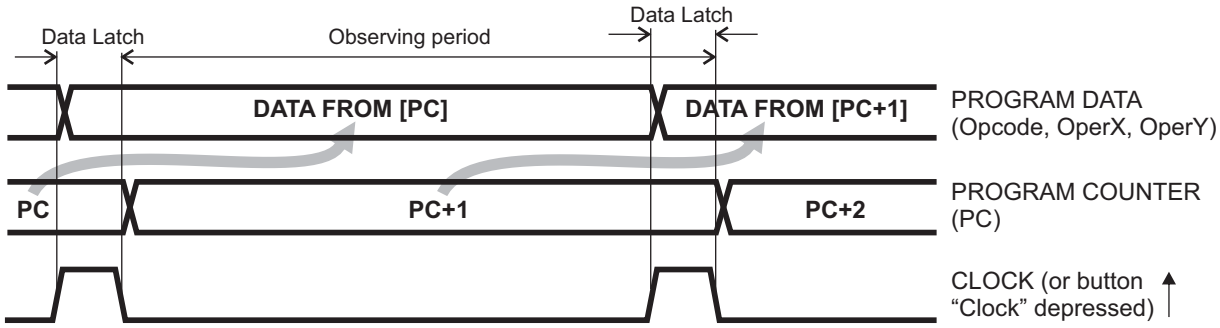
Program Memory Address (or Program Counter, PC) (18)

Program Counter is the **12-bit** pointer which keeps the address of the program word which is read from the **Program Memory** and executed by the processor.

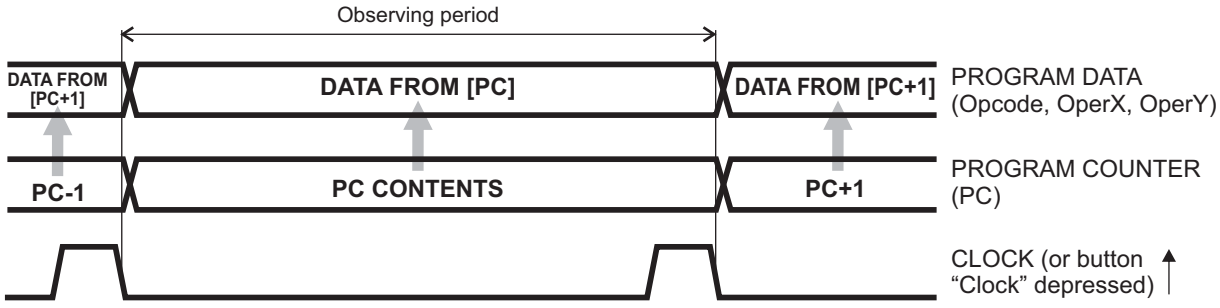
PC indicator (18) always displays the current state of the 12-bit **Program Counter**. The only exception is when the **ALT** button is depressed in the **History submode** of the **Single Step (SS)** mode. In that case (when in **SS** mode, **History** submode, and **ALT** depressed), **PC** indicator displays the 7-bit counter which denotes the depth of the History pointer. In other words, it shows how deep (how many backward steps) you are in the History "time machine" (how many Single Steps backwards you are watching registers and memory contents).

IMPORTANT NOTE: Immediately after reading the current program word from the Program Memory (which is performed automatically at the every clock pulse while the program is running), the **PC** should be automatically incremented by one, always pointing to the **NEXT** instruction, instead of the **CURRENT** one. That means that the **PC** is always ready to read the next instruction and it is a faster way to perform reading. That's why this method is used in normal processors.

Note: the following diagram is simplified. Please note the gray arrows.



That's what happens when we increment **PC** immediately after the instruction was read. If the program is running at a high speed, you wouldn't notice a problem, but in the slow modes (e.g. one instruction in two seconds, or even more in the Single Step (**SS**) mode, the **PC** indicator will always point to the **NEXT** instruction, instead to the **CURRENT** one. This means that you would have the address **[PC+1]** displayed on the **PC** indicator, and the data from the address **[PC]+0** displayed on the **Opcode, Operand X** and **Operand Y** indicators. This could be very confusing, and that's why another method was used in the badge, even if it is not consistent with all other processors:



There is one more consequence of this non-consistency with the **PC** behavior of the real processor. Instruction **JR NN**, which modifies the current PC adding the 8-bit signed value **NN** to the **PC,** uses the **incremented PC value**, although the **current** (non-incremented) **value** is displayed on the **PC** indicator. For instance, if you want to perform the **dead loop** (infinite loop), you should not perform the instruction **JR 0**, but **JR -1**, as the program should jump relatively from the **next** location, and not from the **current** one. Reminder: **-1** is the **signed 2's complement** number, so it should be written as **1111 1111**. Since the Opcode for the instruction **JR** is **1111**, the instruction **JR -1** should be binary coded as **1111 1111 1111**.

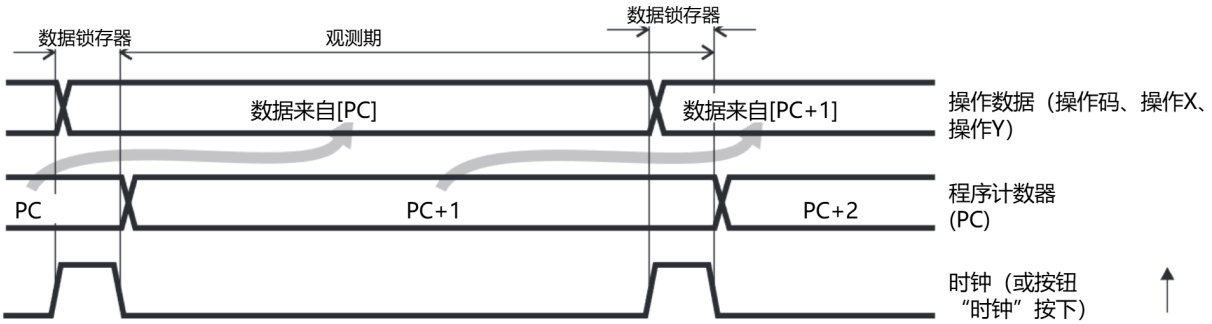
程序存储器地址（或程序计数器，PC）（18）

程序计数器是一个12位指针，它保存从程序存储器中读取并由处理器执行的程序字的地址。

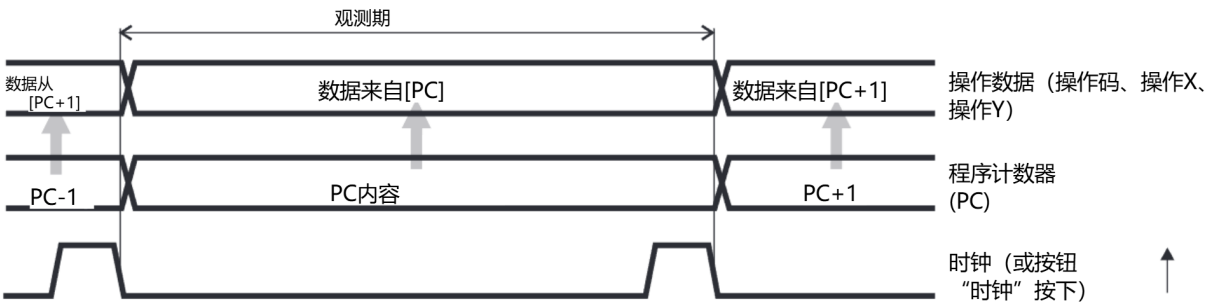
PC指示器（18）始终显示12位程序计数器的当前状态。唯一的例外是在单步（SS）模式的历史记录子模式下按下ALT按钮时。在这种情况下（当处于SS模式、历史记录子模式和ALT按下时），PC指示器显示7位计数器，表示历史记录指针的深度。换句话说，它显示了你在历史“时间机器”中的深度（多少步后退）（你正在观察寄存器和内存内容的多少步后退）。

重要提示：从程序存储器中阅读当前程序字后（程序运行时在每个时钟脉冲自动执行），PC应立即自动递增1，始终指向NEXT指令，而不是CURRENT指令。这意味着PC总是准备好读取下一条指令，这是一种更快的执行阅读的方法。这就是为什么在普通处理器中使用这种方法的原因。

注：下图为简化图。请注意灰色箭头。



这就是当我们在读指令后立即递增PC时发生的情况。如果程序以高速运行，您不会注意到问题，但在慢速模式下（例如，两秒内执行一条指令，或者在单步（SS）模式下执行更多指令），PC指示器将始终指向NEXT指令，而不是CURRENT指令。这意味着您将在PC指示器上显示地址[PC+1]，并在操作码、操作数X和操作数Y指示器上显示来自地址[PC]+0的数据。这可能会非常令人困惑，这就是为什么在徽章中使用另一种方法，即使它与所有其他处理器不一致：

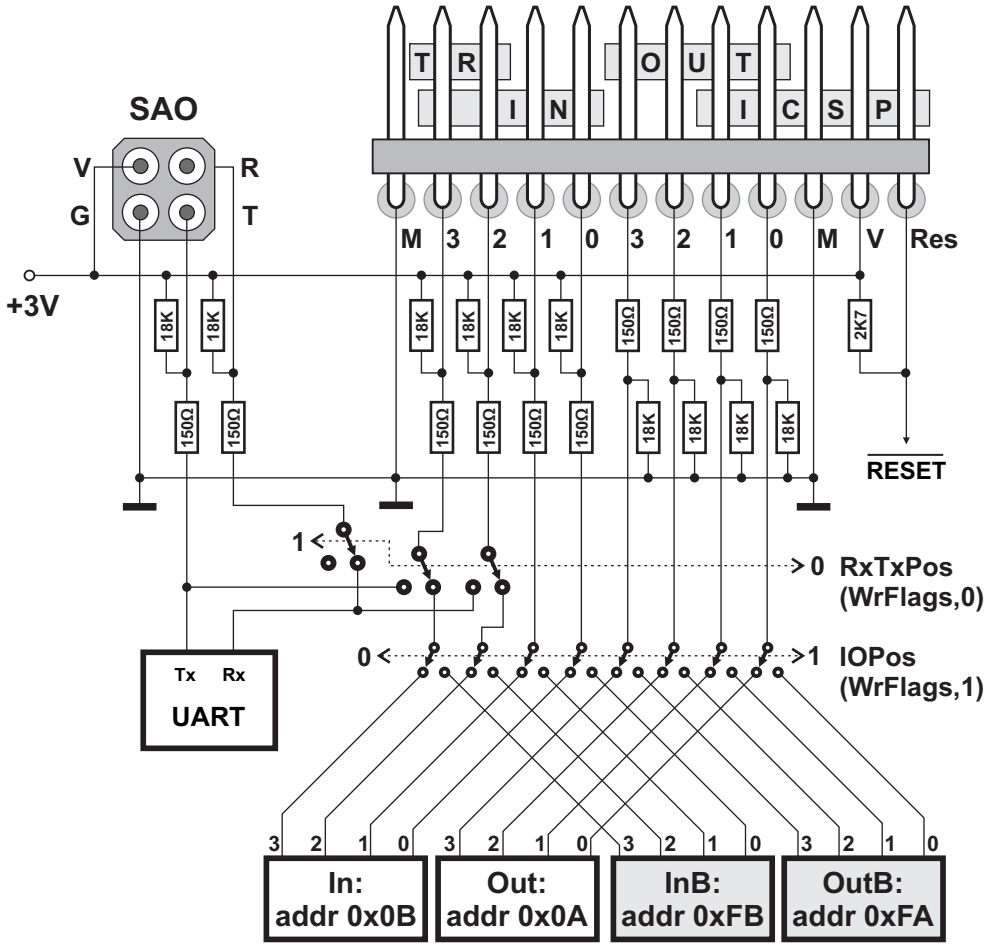


这种与真实的处理器的PC行为不一致还有一个后果。修改当前PC并将8位带符号值NN添加到PC的指令JR NN使用递增的PC值，尽管当前（非递增）值显示在PC指示器上。例如，如果你想执行死循环（无限循环），你不应该执行指令JR 0，而是JR-1，因为程序应该从下一个位置相对跳转，而不是从当前位置跳转。提示：-1是带符号的2的补码，所以应该写为1111 1111。由于指令JR的操作码是1111，所以指令JR-1应该被二进制编码为1111 1111 1111。

Indicators, buttons and connectors

SAO (Shitty Add-On) connector (11)

SAO connector was recently standardized for simple badge add-ons. There are power supply contacts on this connector, but note that there is no **I2C** port, but **UART** terminals **Tx** and **Rx** instead.



Input / Output (I/O) connector (12)

Four **Input** and four **Output** ports are available in the **I/O** connector for hardware expansion. The same connector also offers five rightmost contacts for **ICSP** (**In Circuit Serial Programmer**) which can be used for programming and debugging of the firmware of the unit. Models from **PICKIT** and **ICD** series are pin-to-pin compatible with the connector.

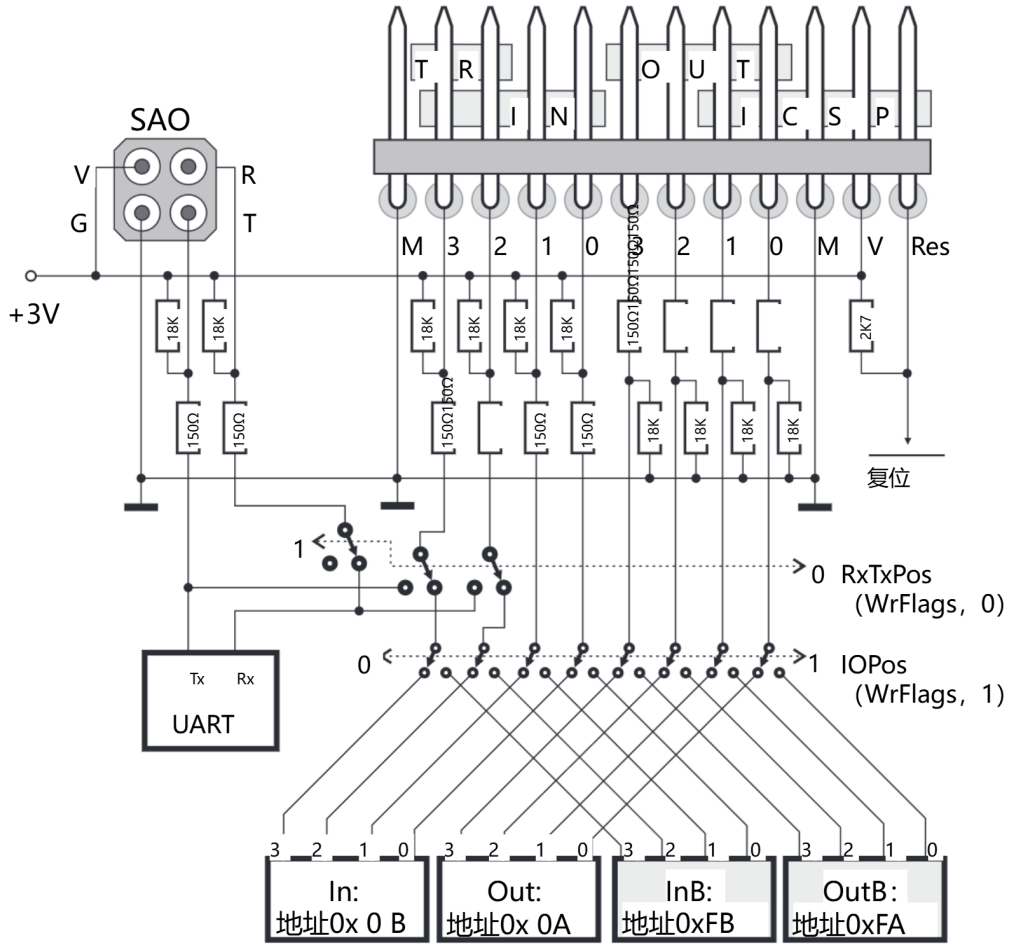
There are two bits in the **SFR** register **WrFlag** which control the pin commutations of the **I/O** connector. When the first one, **RxTxPos (WrFlags,0)**, is set, **UART** is internally connected to the pins of the **I/O** connector (**Tx** is also available on the **SAO** connector, with the same signal and can be used in parallel with **Tx** signal on **I/O** connector). When the first one, **RxTxPos (WrFlags,0)**, is set, port inputs **3** and **2** read the **Tx** and **Rx** states and can not be used as general purpose **I/O** pins.

Bit **IOPos (WrFlags,1)** decides which registers will be used as **Output** and **Onput** registers. Default registers are **R10** and **R11** in the **General Purpose** group, on page **0x00** of the **Data Memory** (locations **0xFA** and **0xFB**). If this bit is set, all Inputs and Outputs are redirected to the **SFR** group, on page **0x0F** of the **Data Memory** (locations **0xFA** and **0xFB**), so registers **R10** and **R11** can be used as **General Purpose** registers.

By default, **RxTxPos** and **IOPos** are reset. Default state is initialized at every Program **RUN** and **Break**. Note that Program **SAVE** and **LOAD** functions always use **Tx** and **Rx** pins on the **I/O** connector (not on **SAO** connector), regardless of the bit **RxTxPos** state.

SAO（Shitty Add-On）连接器（11）

SAO连接器最近被标准化用于简单的徽章附加组件。此连接器上有电源触点，但请注意，没有I2C端口，而是连接Tx和Rx端子。



输入/输出（I/O）连接器（12）

I/O连接器中有四个输入和四个输出端口，可用于硬件扩展。同一个连接器还为ICSP（在线串行编程器）提供了五个最右边的触点，可用于编程和调试单元的固件。PICKIT和ICD系列的型号与连接器引脚兼容。

SFR寄存器Flag中有两位控制I/O连接器的引脚交换。当第一个RxTxPos（RxFlags，0）被设置时，RxTxPos内部连接到I/O连接器的引脚（Tx也可用于SAO连接器，具有相同的信号，并可与I/O连接器上的Tx信号并行使用）。当第一个RxTxPos（RxFlags，0）被设置时，端口输入3和2读取Tx和Rx状态，不能用作通用I/O引脚。

位IOPos（寄存器标志，1）决定哪些寄存器将用作输出和输入寄存器。
默认寄存器为通用组中的R10和R11，位于数据存储器的第0x 00页（位置0xFA和0xFB）。如果该位置1，则所有输入和输出均重定向至数据存储器页面0x 0 F（位置0xFA和0xFB）上的SFR组，因此寄存器R10和R11可用作通用寄存器。

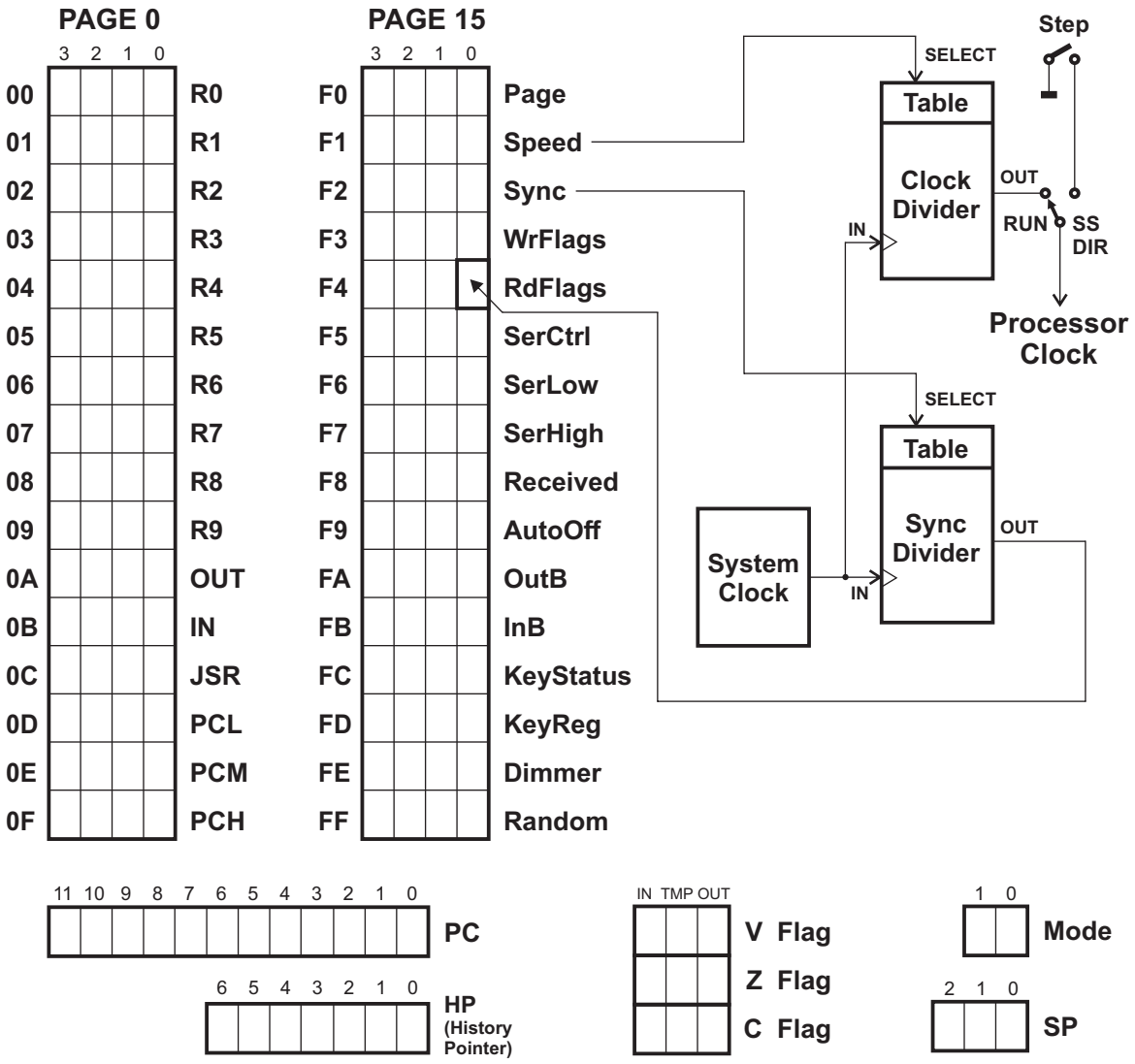
默认情况下，重置RxTxPos和IOPos。默认状态在每次程序运行和中断时初始化。请注意，无论位RxTxPos状态如何，程序保存和加载功能始终使用I/O连接器（而不是SAO连接器）上的Tx和Rx引脚。

Programmer's model

This is the hypothetical **4-bit** processor which is simulated by the firmware executed on the **16-bit** microcontroller **PI24FJ256GA704**.

There is the **256-nibble** (**256×4**) **Data Memory**, which can not be expanded. Program is executed from the **4K** words (**4096×12**) **Program Memory**.

There are a total of **16 Main Registers** on **Page 0** (**0x00-0x0F**) of the **Data Memory**. Some of them (Registers **R0-R9**) are **General Purpose Registers**, and the rest of them (Registers **R10-R15**) are **Special Purpose Registers**, as they are dedicated to specific functions. This Register set can be reconfigured, so that two of the registers (**OUT** register, **R10** and **IN** register, **R11**) are moved to the **Special Function Register** group at **Page 15**, making free space for two more **General Purpose Registers**.



There are also a total of **16 Special Function Registers** (**SFR**), which are used to control processor's pseudo-hardware and perform special operations. All **Special Purpose Registers** are located on the **Page 0xF** (**0xF0-0xFF**) of the **Data Memory**. These registers are described in the manual **Special Function Registers**.

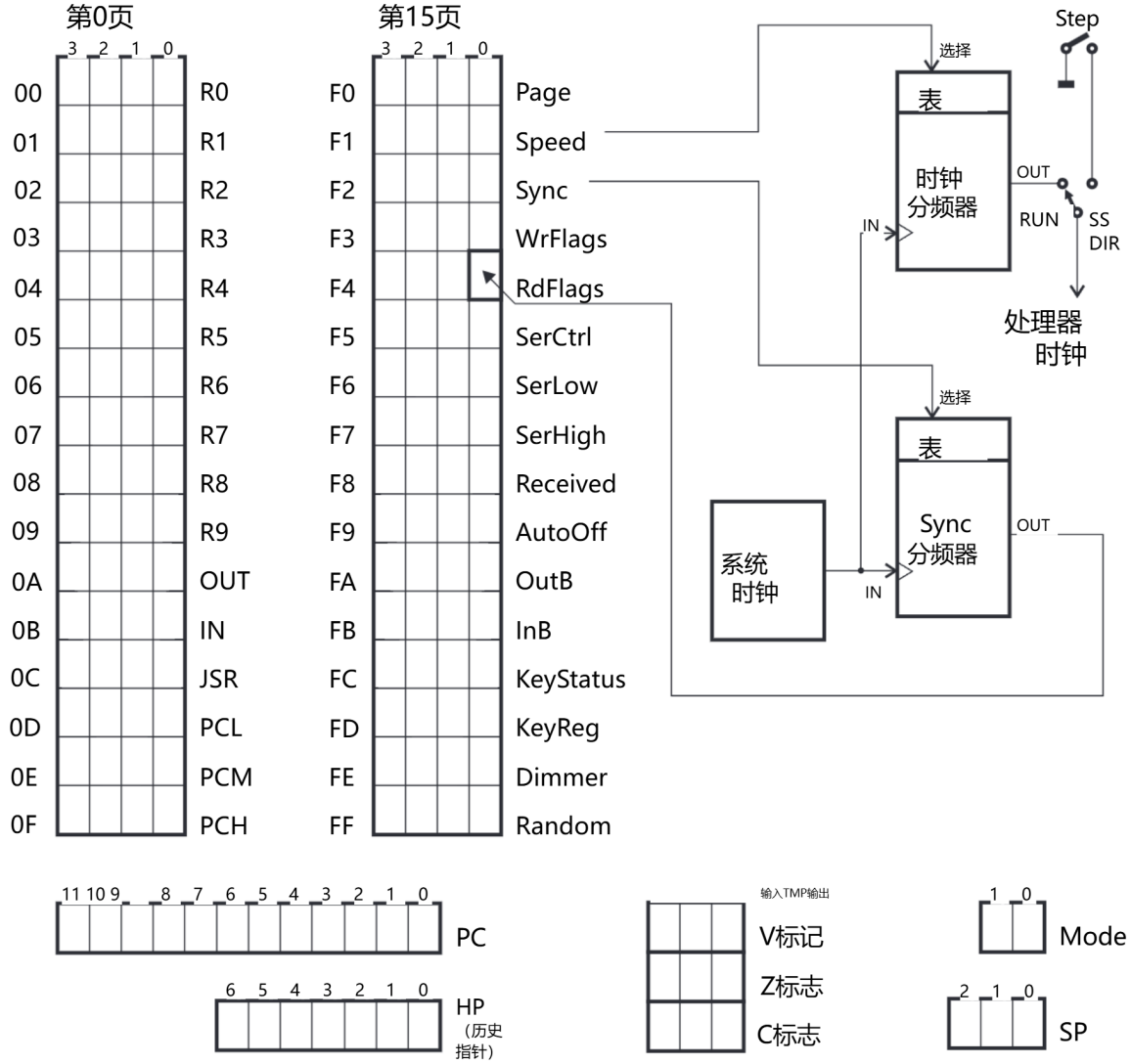
Stack Pointer (**SP**) is a **3-bit** register, not accessible directly to the user. **SP** memory is on the **Page 1 (0x10-0x1E)** of the **Data Memory**. It occupies **15** nibbles, which is enough for **5** levels of subroutines. The last nibble (**0x1F**) is not used, so it is available to the programmer. Also, if the programmer is sure that not all Stack levels will be used in the program, nibbles from that area can be used as a **General Purpose Data Memory**. Note that there are no **PUSH** or **POP** instructions.

程序员模型

这是假设的4位处理器，由16位微控制器PI 24 FJ 256 GA 704上执行的固件模拟。

有256半字节（256×4）数据存储器，不能扩展。程序从4K字（4096×12）程序存储器中执行。

数据存储器的第0页（0x 00 - 0x 0 F）上共有16个主寄存器。一些
其中（寄存器R 0-R9）是通用寄存器，其余（寄存器R10-R15）是专用寄存器，因为它们专用于特定功能。可以重新配置该寄存器组，以便将其中两个寄存器（OUT寄存器R10和IN寄存器R11）移至第15页的特殊功能寄存器组，从而为另外两个通用寄存器腾出空间。



还有总共16个特殊功能寄存器（SFR），用于控制处理器的伪硬件和执行特殊操作。所有专用寄存器均位于数据存储器的页面0xF（0xF 0 -0xFF）。手册《特殊功能寄存器》中介绍了这些寄存器。

堆栈指针（SP）是一个3位寄存器，用户不能直接访问。SP内存位于数据内存的第1页（0x 10 - 0x 1 E）。它占用了15个字节，这对于5个级别的子程序来说是足够的。最后一个半字节（0x 1F）未被使用，因此可供程序员使用。此外，如果程序员确定程序中不会使用所有堆栈级别，则该区域的半字节可用作通用数据存储器。请注意，没有PUSH或POP指令。

Programmer's model

Program Counter (**PC**) keeps the **Program Memory Address**. As the **Program Memory** occupies **4096** words, **PC** register has **12** bits.

All internal registers and many internal logic states of the processor are displayed in real-time. Also, two selected pages (**16** nibbles each) are displayed on the LED matrix. However, all LED indicators are multiplexed, so some fast processor operations are subject to interference effect.

Internal clock frequency is user-selectable (and also selectable by the program) in **15** steps from **0.5 Hz** to **100 KHz**, and the **16th** clock frequency step is at the maximum speed, which is about **250 KHz**, but not guaranteed and not synchronized to the internal time base. Every instruction is executed in one clock cycle, so the maximum execution speed is about **0.25 MIPS**.

Another user-selectable (also selectable by the program) is the **Sync Timer.** It generates the internal heartbeat stream which has no impact neither on the hardware nor on the system firmware, but only sets the single bit in the **SFR** group (bit **0**, named **UserSync** in **RdFlags** register). User's program can test this bit as a flag at the handshaking manner, and thus synchronize the program flow to the uniform time periods. Available periods are arranged in **16 steps** from **1 ms** to **1 sec**.

There are three flags: **Overflow (V), Zero (Z)** and **Carry (C)**. Adder/Subtractor contains 4 Full Adder circuits, with Carry logic and Data inverters which enable full Add/Subtract operations for **unsigned** and **signed 4-bit** numbers with **Carry**.

Note: According to the schematic, data inverter for D0-D3 inverts the Destination bits only. However, in most cases it inverts Source bits, but there was no space for these bits here.

Every instruction has the length of **12 bits**. There is a table with all instructrions in this manual, and also the detailed description of every instruction with examples.

There are no **Long Jump** or **Call** instructions, so **Long Jumps** and **Subroutine Calls/Returns** are performed by writing a nibble in the Registers which are in the **Page 0**: writing to the register **PCL** performs the **Long Jump** to the location addressed by registers **PCL (0x13)**, **PCM (0x14)** and **PCH (0x15)**. Registers **PCM** and **PCH** must be pre-loaded with the desired high portion **(bits 11-4)** of the address.

Subroutine Call is performed in the similar manner, when the four lowest bits are written to the register **JSR (0x12)**. Registers **PCM** and **PCH** should be pre-loaded also. When the subroutine is called, the return address is written to the **Page 1** (**0x10-0X1E**) of the Data Memory, using the SP register, and then the SP register is incremented by 1. Address where the **12-bit Return Address** is stored, is calculated by the formula **0x10+(3×[SP])**.

When the **RETURN** instruction is executed, the process is reversed: Return address is read from the **SP** calculated address to the **PC,** and the **SP** is decremented. Also, **4-bit** literal number is loaded to the **R0** (which is on the address **0x00**). This can be used for lookup table read.

Not all instructions will initiate **Long Jumps** or **Calls** by writing to the **PCL** and **JSR** registers. Only the following instructions, which point to registers **PCL** or **JSR** as the destination, will do that:

MOV	**RX, RY**	(for calculated Jumps and Calls)
MOV	**RX,N**	(for simple Jumps and Calls)
INC	**RY**	(for repeated Table Reads)
DEC	**RY**	(for repeated Table Reads in reverse order)

MODES OF OPERATION

As it was mentioned before, there are four basic modes:

DIR	(Register addressing, directly executed instructions without memory usage)
SS	(Single stepping of the program from the Program Memory, temporarily editable)
RUN	(Program execution)
PGM	(Program writing or editing)

Mode **SS** also contains the submode **HISTORY**, which enables reviewing and analyzing of the last **127** program steps, with all processor signals and Data Memory contents.

There are also two special modes, used for hardware and firmware maintenance: **Test Mode** and **Bootload Mode**.

程序员模型

程序计数器（PC）保存程序存储器地址。由于程序存储器占用4096个字，PC寄存器有12位。

处理器的所有内部寄存器和许多内部逻辑状态都可以实时显示。此外，两个选定的页面（每个页面16个半字节）显示在LED矩阵上。但是，所有LED指示灯都是多路复用的，因此一些快速处理器操作会受到干扰影响。

内部时钟频率是用户可选择的（也可由程序选择），从0.5 Hz到100 KHz，15个步长，第16个时钟频率步长为最大速度，约为250 KHz，但不保证与内部时基同步。每个指令在一个时钟周期内执行，因此最大执行速度约为0.25 MIPS。

另一个用户可选择的（也可由程序选择）是同步定时器。它生成内部心跳流，对硬件和系统固件都没有影响，但只设置SFR组中的单个位（位0，RdFlags寄存器中名为UserSync）。用户的程序可以在握手方式下将该位作为标志进行测试，从而将程序流同步到统一的时间段。可用的周期以16个步长排列，从1 ms到1 sec。

有三个标志：溢出（V），零（Z）和进位（C）。加法器/减法器包含4个全加器电路，带有进位逻辑和数据反相器，可实现带进位的无符号和有符号4位数的全加减法操作。

注：根据原理图，D 0-D3的数据反相器仅反相目标位。
然而，在大多数情况下，它会反转源位，但这里没有这些位的空间。

每条指令的长度为12位。本手册中有一个表格，列出了所有指令，并对每个指令进行了详细说明，并附有示例。

没有长跳转或调用指令，因此长跳转和子程序调用/返回是通过在页0中的寄存器中写入半字节来执行的：写入寄存器PCL执行长跳转到寄存器PCL（0x 13）、PCM（0x 14）和PCH（0x 15）寻址的位置。寄存器PCM和PCH必须预先加载所需的地址高位部分（位11-4）。

当四个最低位被写入寄存器JSR（0x 12）时，以类似的方式执行子例程Call。寄存器PCM和PCH也应预先加载。调用子例程时，返回地址使用SP寄存器写入数据存储器的第1页（0x 10 - 0X 1 E），然后SP寄存器递增1。存储12位返回地址的地址由公式0x 10+（3×[SP]）计算。

当执行RETURN指令时，过程相反：返回地址从SP计算的地址读取到PC，SP递减。此外，4位文字数被加载到R0（位于地址0x00上）。这可以用于查找表读取。

不是所有的指令都会通过写入PCL和JSR寄存器来启动长跳转或调用。只有下面的指令，指向PCL或JSR作为目的地，才能做到这一点：

```
MOV RX，RY（用于计算的跳转和调用）
MOV RX，N     (for简单的跳跃和呼叫)
INC RY        (for重复表读取)
DEC   RY      (for以相反顺序重复表读取)
```

操作模式

如前所述，有四种基本模式：

```
DIR     (寄存器寻址，直接执行指令而不使用内存)
SS      (从程序存储器单步执行程序，暂时可编辑)
RUN     (程序执行)
PGM     (程序编写或编辑)
```

模式SS还包含子模式HISTORY，它允许查看和分析最后127个程序步骤，以及所有处理器信号和数据存储器内容。

还有两种特殊模式，用于硬件和固件维护：测试模式和引导加载模式。

Programmer's model

ERROR PROCESSING

The only **Fatal Errors** which are possible at runtime, are the **Stack Errors**. Stack **Underflow** and Stack **Overflow** cause the unconditional program termination, with Stack indicator blinking, showing the illegally attempted state **110 (6)** on overflow, or **111 (-1)** on Underflow

INSTRUCTION SET

There is a total of **31** instructions, which are listed and described in detail in the manual **INSTRUCTION SET**. Only **11** instructions are available in **DIR** mode (manual **INSTRUCTION SET IN DIRECT MODE**).

Opcode and operand fields are divided in three **4-bit** groups, so that there are **4-bit** and **8-bit** opcodes (bits **11-4** or **7-4**) with one or two operands (only in a few special cases, used for **skip** and **bit manipulations**, there are **8-bit** opcodes and two **2-bit** operands). This strict and clear **Opcode/Operand** allocation inside the **12-bit Program Word space** greatly eases writing of programs in the pure machine language. Also, there are **4-to-16** decoders with LED indicators, which interactively display the program code disassembled in some way, making it much easier to read and write, without learning the instruction codes.

After the Data Memory Organization on the next page, there is a table with all instructions listed. After that, the detailed description of every instruction follows, with examples, coding schemes and flags affected.

DATA MEMORY

Data memory contains **256** nibbles (**256×4**), organized in **16** pages. Page **0 (0x00-0x0F)** contains a total of **16** main registers. The first **10** registers **(R0-R9)** are the General Purpose Registers, and the remaining **six (0x0A-0x0F)** are the **Special Function Registers**. Two functions (**Out** and **In**) which regularly occupy locations **0x0A** and **0x0B**, can be redirected to the **SFR** area on the last page, which allows free access to 12 General Purpose Registers. In that case, **Out** and **In** registers are at locations **0xFA** and **0xFB**.

The second page of the **Data Memory** is assigned to the **Stack.** It is the area where the **Return Address** will be written at every **Subroutine Call**. A total off **5** levels of **Stack** is allowed, and every **Return Address** takes **12** bits (**4** nibbles), so a total of **15** nibbles can be used for the **Stack**.

Page **14** can be used as the **Shaddow Register** area for a selected number of **Main Registers** from the page **0.** Instruction **EXR N** swaps the contents of pages **0** and **14**, in the length defined in the literal number **N**.

Page **15** is the Special Function Register (**SFR**) area, which contain **16** different registers with special functins. There is a detailed description of **SFR** in the manual "**Special Function Registers**".

Every portion of **Data Memory**, in the length of two pages (**32** nibbles), can be interactively displayed on LED matrix. Register **PAGE** (which is in the **SFR** area at the address **0xF0**) determines which page will be displayed. The selected page is on the right halve of the display, and the next page is on the left. If the selected page is **15**, then the next page, displayed on the left halve, will be **0**.

程序员模型

错误处理

在运行时可能出现的唯一致命错误是堆栈错误。堆栈下溢和堆栈上溢导致无条件程序终止，堆栈指示器闪烁，表示上溢时非法尝试状态110（6），下溢时非法尝试状态111（-1

指令集

总共有31个指令，在手册INDUSTRY SET中列出并详细描述。只有11个指令可在直接模式下使用（手动直接设置）。

操作码和操作数字段分为三个4位组，因此存在具有一个或两个操作数的4位和8位操作码（位11 - 4或7 - 4）（仅在少数特殊情况下，用于跳过和位操作，存在8位操作码和两个2位操作数）。在12位程序字空间内的这种严格和明确的操作码/操作数分配大大简化了纯机器语言的程序编写。此外，还有4到16个带有LED指示灯的解码器，它们以某种方式交互式地显示分解的程序代码，使其更容易阅读和编写，而无需学习指令代码。

在下一页的数据存储器组织之后，有一个列出所有指令的表。在此之后，每个指令的详细描述如下，与例子，编码方案和标志的影响。

数据存储器

数据存储器包含256个半字节（256×4），分为16页。页面0（0x 00 - 0x 0 F）包含总共16个主寄存器。前10个寄存器（R 0-R9）是通用寄存器，其余6个（0x 0A-0x 0 F）是特殊功能寄存器。两个函数（输出和输入）经常占用位置0x 0A和0x 0 B，可以重定向到最后一页上的SFR区域，该区域允许自由访问12个通用寄存器。在这种情况下，Out和In寄存器位于位置0xFA和0xFB。

数据存储器的第二页被分配给堆栈。它是在每次子程序调用时写入返回地址的区域。允许总共5级堆栈，每个返回地址占用12位（4个半字节），因此总共15个半字节可用于堆栈。

第14页可用作从第0页起选定数量的主寄存器的阴影寄存器区域。指令EXR N以文字数N中定义的长度交换页0和14的内容。

第15页是特殊功能寄存器（SFR）区域，其中包含16个具有特殊功能不同寄存器。在手册“特殊功能寄存器”中有SFR的详细说明。

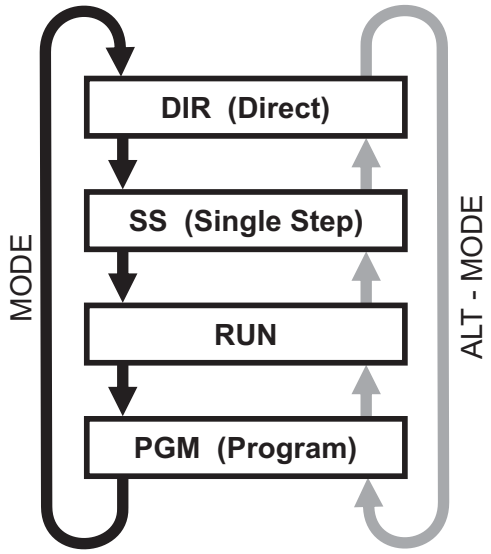
数据存储器的每一部分，在两页（32半字节）的长度，可以交互式显示在LED矩阵。寄存器PAGE（位于SFR区域的地址0xF0处）决定显示哪一页。所选页面位于显示屏的右半部分，下一页位于左侧。如果所选页面为15，则显示在左半部分的下一页将为0。

Direct (DIR) Mode

MODE button and indicators (16)

There are four main modes: **DIR** (Direct), **SS** (Single Step), **RUN** and **PGM** (Program) mode. They are selected sequentially by pressing the **Mode** button. At every press, the next mode is set in increasing order. When **ALT** button is depressed, **Mode** button selects the mode in reversed order.

Every button from the **Command Group** (17) has the different function, which is depended on the current **Mode**. The description of available modes is on the following pages.

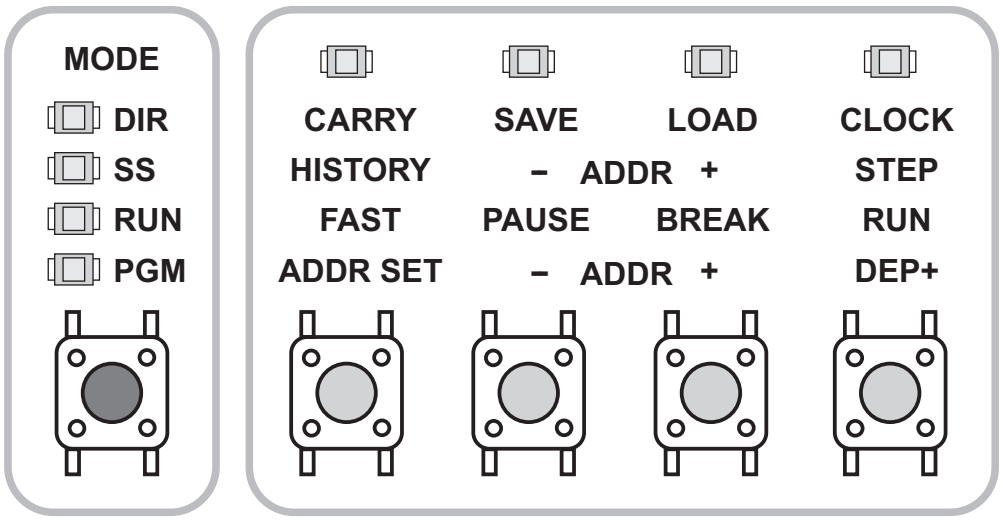


Direct writing and execution of instructions using **Register** addressing. Also program **Save** and **Load**.

Program execution from the **Program Memory** with manual modification and single stepping.

Program execution from the **Program Memory** driven by the internal adjustable **Clock**.

Program editor for direct writing to the internal **Program Memory**

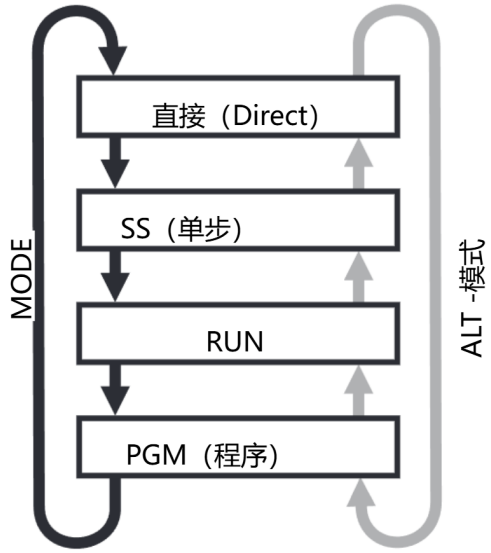


直接（直接）模式

模式按钮和指示灯（16）

有四种主要模式：直接（Direct）、单步（Single Step）、运行（RUN）和程序（PGM）模式。通过按下模式按钮，可依次选择它们。每次按下时，下一个模式以递增顺序设置。当按下ALT按钮时，模式按钮以相反的顺序选择模式。

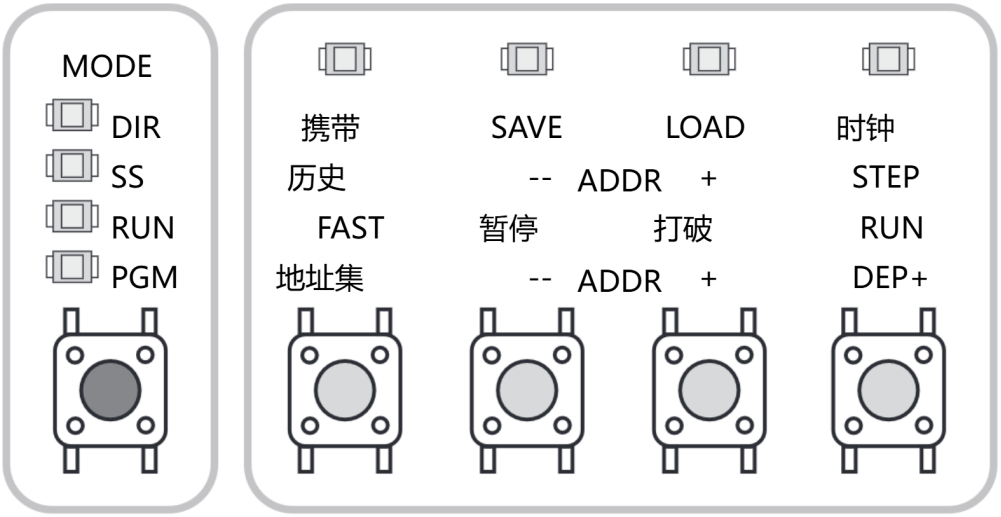
命令组（17）中的每个按钮具有不同的功能，这取决于当前模式。可用模式的说明见以下页面。



使用寄存器寻址直接写入和执行指令。还可编程保存和加载。

从程序存储器执行程序，手动修改和单步执行。
程序执行从程序存储器驱动的内部可调时钟。

程序编辑器，用于直接写入内部
程序存储器



Direct (DIR) Mode

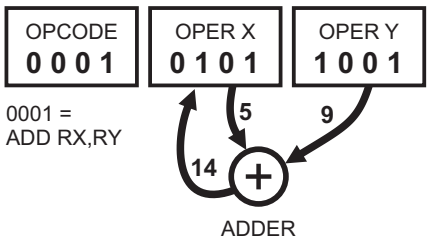
DIR Mode can be used for experimenting with different parts of the processor core: **Accumulator**, **Register X**, **Register Y**, **Adder/Subtractor**, **Logic Group**, **Flags** and so on. Most instructions can be executed by pressing the button **Clock**, but there is no indirect addressing mode, as registers **Operand X** and **Operand Y** contain only literal values. Data Memory is not accessible, which also means that General Function registers and Special Function registers do not exist in **DIR** mode. **I/O** connector is also not accessible. Programs written in **Program Memory** have no effect in **DIR** mode. The same is valid for **Program Counter (PC)**, **Stack Pointer** or **Page** registers.

All flags are normally active in the **DIR** mode.

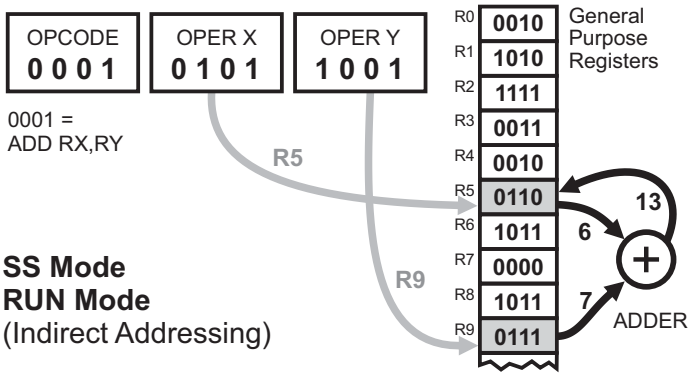
Instructions which communicate with the Data Memory cannot be executed in **DIR** mode. If the non-existing instruction is selected, **LED** which points to the instruction in the vertical decoded fields, will blink and the **Clock** button will not be active.

Registers **RX** (available in the **Operand X** field) and **RY** (in **Operand Y** field) are used as the two directly accessible 4-bit registers, and they do not point to the first page of the Data Memory, like in all other modes. If register **RX** or **RY** is the destination, the result will be written directly to the register **RX** or **RY**.

Here's an example: if the instruction **ADD RX,RY** (**0001**) is selected in the **Opcode** field, in the **DIR** mode it will be executed so that value displayed in the **Operand Y** field is added to the value displayed in the **Operand X** field and, when the button **Clock** is pressed once, the result (which is indicated on the **Adder** output and the **Accumulator** input) will be written to the register **RX** (indicators in the **Operand X** field).



Direct Mode
(Register Addressing)



SS Mode
RUN Mode
(Indirect Addressing)

In **SS** and **RUN** modes, processor will operate with contents of the General Purpose registers in **Data Memory**, which are addressed with **RX** and **RY** registers. For instance, if the value of **Operand X** is **0101** (decimal **5**) and the value of **Operand Y** is **1001** (decimal 9), processor will read values of registers **R5** and **R9**, which are in the **Data Memory** at the addresses **0x05** and **0x09**, adder will calculate the sum and write the result to the register **R5** (**Data Memory** address **0x05**).

User Program can be saved or loaded in **DIR** mode. Media available for program storing is the internal **Flash** memory, which can store up to **15** user's programs, or the external store unit which communicates through **UART** (**Universal Asynchronous Receiver-Transmitter**). Transmit and Receive terminals are with **3V** logic levels, so the best way to implement the serial communication with the laptop or desktop computer is the **USB/Serial** converter. Don't forget to cross Rx and Tx conductors, but also to **leave the + terminal from the converter unconnected**! Two power sources should never be connected in parallel, as the consequences could be unpredictable.

The same connection with the computer enables program sharing and loading of externally generated programs (assembler or some other way to generate the program code).

Although the **Display Matrix** is not active in **DIR** mode, there are two exceptions. The first one is displaying of **Program Version/Revision** and **Release Da**te, which is performed after the **Master Reset** (by removing and reinserting batteries or shorting pins **Res** and **G** on the **I/O** connector). This will be displayed only once after the Master Reset, and it will be cleared when any button is pressed.

直接（直接）模式

直接（直接）模式

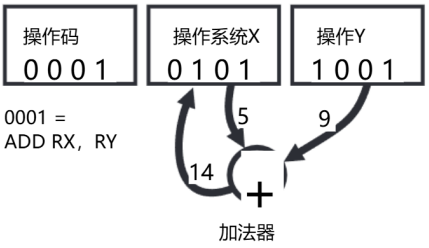
测试模式可用于测试处理器内核的不同部分：
累加器、寄存器X、寄存器Y、加法器/减法器、逻辑组、标志等。大多数
指令可以通过按下时钟按钮来执行，但没有间接寻址模式，因为寄存器操作数X和操作数Y只包含文字值。数据存储器不可访问，这也意味着通用功能寄存器和特殊功能寄存器在EEPROM模式下不存在。I/O接口也不可访问。写入程序存储器的程序在脱机模式下无效。这同样适用于程序计数器（PC）、堆栈指针或页寄存器。

所有的标志通常都是活跃的，在该模式。

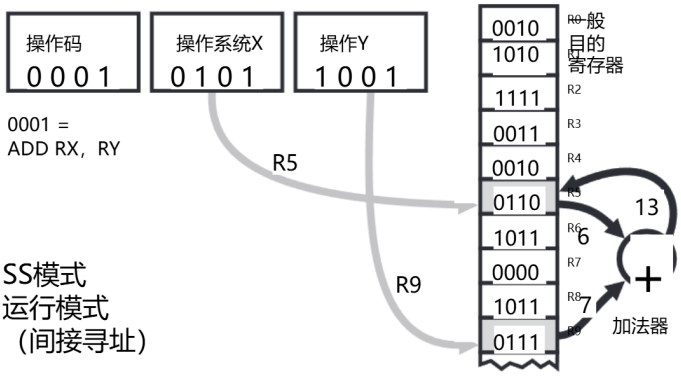
与数据存储器通信的指令不能在同步模式下执行。如果选择了不存在的指令，则指向垂直解码字段中的指令的LED将闪烁，并且时钟按钮将不活动。

寄存器RX（在操作数X字段中可用）和RY（在操作数Y字段中）用作
两个可直接访问的4位寄存器，它们不指向数据存储器的第一页，就像所有其他模式一样。如果寄存器RX或RY是目的地，则结果将直接写入寄存器RX或RY。

下面是一个示例：如果在操作码字段中选择了指令ADD RX，RY（0001），则在计数器模式下将执行该指令，以便将操作数Y字段中显示的值与操作数X字段中显示的值相加，并且当按下时钟按钮一次时，结果（在加法器输出和累加器输入上显示）将写入寄存器RX（操作数X字段中的指示符）。



直接模式
（寄存器寻址）



SS模式
运行模式
（间接寻址）

在SS和RUN模式下，处理器将使用数据存储器中通用寄存器的内容进行操作，这些寄存器通过RX和RY寄存器进行寻址。例如，如果操作数X的值是0101（十进制5），操作数Y的值是1001（十进制9），处理器将读取寄存器R5和R9的值，它们位于数据存储器中的地址0x 05和0x 09，加法器将计算和并将结果写入寄存器R5（数据存储器地址0x 05）。

用户程序可以在脱机模式下保存或加载。可用于程序存储的介质是内部闪存，可存储多达15个用户程序，或通过UART（通用异步接收器-发送器）进行通信的外部存储单元。发射和接收端子具有3V逻辑电平，因此实现与笔记本电脑或台式电脑串行通信的最佳方式是USB/串行转换器。不要忘记交叉Rx和Tx导体，但也要保持转换器的+端子未连接！两个电源不应并联，因为后果可能无法预测。

与计算机的相同连接使程序共享和加载外部生成的程序（汇编程序或其他方式生成程序代码）成为可能。

尽管显示矩阵在DIR模式下不处于活动状态，但有两个例外。第一个是显示程序版本/修订版和发布日期，这是在主复位后执行的（通过移除和重新插入电池或短路I/O连接器上的Res和G引脚）。这将只显示一次后，主复位，它将被清除时，按下任何按钮。

Direct (DIR) Mode

Another exception of the **Display Matrix** function in **DIR** mode is displaying the **Flash** occupancy before the program saving or loading. To see **Flash** occupancy in **DIR** mode, just keep the **ALT** button depressed. Every pixel in this case represents the occupancy of **512** program words.

	Carry	Save	Load	Clock
	Toggle Carry Flag	Send Program Memory to Serial Port	Load Program Memory from Serial Port	Master Clock source
ALT		Save Program Memory to selected Flash	Load Program Memory from selected Flash	

	Opcode	Operand X	Operand Y	Data In
	Instruction Opcode (bits 11-8)	Direct Operand X or Opcode (bits 7-4)	Direct Operand Y (bits 3-0)	Toggle between BINary and SELect mode
ALT	Dimmer level select	Baud Rate select	Flash portion select for Save / Load	

DIR Mode: Button CARRY

Indicator **Carry** in the **Command Group** (17) is automatically updated at every instruction execution, but user can toggle it by pressing the button **Carry** under the indicator. It can be helpful in experimenting and watching how **Carry Flag** affects execution in certain processor core sections, especially in **Adder/Subtractor**.

DIR Mode: Button SAVE

Saving to the external unit through the Serial Port is performed by simply pressing **SAVE** button in **DIR** mode.

Note that the unit has no feedback from the external unit, and it will send the program immediately to the **Serial Port**, even if there is no connection. So it is essential that everything is prepared before the button **SAVE** is pressed, and that the external unit already expects data from the port. This is valid also if the program is shared directly between the two equal units.

Here are the rules for serial **Save/Load**:

- Serial Port settings: **9600, N, 8, 1** (not affected by **SFR** setting in **SerCtrl** register)
- Hardware considerations: **3V CMOS** Logic levels (**Tx/Rx** on **I/O connector** only)
- **Tx/Rx** on **SAO connector** can not be used for program saving or loading
- Software protocol (in Hexadecimal form):

1. Header **6** bytes:	**00 FF 00 FF A5 C3**
2. Program length **2 bytes** (in 16-bit words, Low byte first):	**NN NN**
3. Program **NN 0N×Program Length** (Low first):	**NN 0N, NN 0N, NN 0N...**
4. 16-bit Checksum **2 bytes** (items 2 and 3 only, Low first):	**NN NN**

Since the program data is contained of **12-bit** words, writing format is **16 bits** for every word, but the upper nibble (bits **15-12**) is always dummy **0**. Header is a simple 6-byte string, and all other items are 16-bit numbers in **Little Endian** order (Least Significant Byte first).

直接（直接）模式

另一个例外的显示矩阵功能在显示模式是显示闪存占用前的程序保存或加载。要查看Flash占用率，只需按住ALT按钮。在这种情况下，每个像素表示512个程序字的占用。

	携带	Save	Load	时钟
	切换 进位标志	发送音色 存储器以 串行端口	加载程序 内存从 串行端口	硕士 时钟源
ALT		Save程序 存储器以 选择的闪存	加载程序 内存从 选择的闪存	

	操作码	操作数X	操作数Y	数据
	指令 操作码 (bits（第11-8段)	直接 操作数X或 操作码（位7-4）	直接 操作数Y (bits 3-0)	之间切换 二进制和 选择模式
ALT	调光 层级选择	波特率 选择	毛边部分 select for 保存/加载	

操作模式：按钮CARRY

指令组（17）中的进位指示器在每次指令时自动更新
执行，但用户可以通过按下指示器下的Carry按钮进行切换。它可以帮助实验和观察进位标志如何影响某些处理器核心部分的执行，特别是在加法器/减法器中。

操作模式：按钮保存

通过串行端口保存到外部单元只需在USB模式下按下保存按钮即可执行。

请注意，该单元没有来自外部单元的反馈，即使没有连接，它也会立即将程序发送到串行端口。因此，在按下“保存”按钮之前，必须准备好一切，并且外部单元已经期望来自端口的数据。如果程序直接在两个相等的单元之间共享，这也是有效的。

以下是串行保存/加载的规则：

- 串行端口设置：9600、N、8、1（不受SerCtrl寄存器中SFR设置的影响）
- 硬件注意事项：3V CMOS逻辑电平（仅限I/O连接器上的Tx/Rx）-SAO连接器上的Tx/Rx不能用于程序保存或加载-软件协议（十六进制格式）：

1. 标题6字节：00 FF 00 FF A5 C3
2. 程序长度2个字节（16位字，低字节优先）：NN NN
3. 程序NN 0 N ×程序长度（低优先）：NN 0 N，NN 0 N，NN 0 N...
4. 16-位校验和2字节（仅项目2和3，低电平优先）：NN NN

由于程序数据包含12位字，所以写入格式为每个字16位，但上半字节（位15-12）总是伪0。Header是一个简单的6字节字符串，所有其他项都是16位数字，按小端顺序排列（最低有效字节优先）。

However, if there is nothing to save (if the whole Program Memory is empty), indicator **SAVE** will blink and no data will be sent to the Serial Port. In that case, any button will quit blinking and return the unit to the previous state.

During program saving, the taskbar appears on the three vertical decoded indicator bars.

If the program has to be saved to the internal **Flash** memory, **ALT-SAVE** has to be typed (hold **ALT** while pressing **SAVE**). But before that, a few steps should be performed. First, select the address of the portion of Flash memory where the program will be saved. There are a total of **15** available portions (the 16th one has a special function), and the selection of the desired one should be performed by pressing and holding the **ALT** button, while selecting the portion by pressing buttons in the **Operand Y** field. The helpful feature is that the Flash occupancy is displayed on the Display Matrix, so you just have to match the vertical indicator bar next to the desired portion. When you are finished with the Flash portion choice, just press **SAVE** while you are still holding **ALT**.

Writing to the Flash memory is performed much faster than writing to the **Serial Port**, so no taskbar is displayed here. As the master processor, which drives the unit, halts the program execution during programming, display matrix refresh is inhibited and all LEDs (except **SAVE**) are switched off for a fraction of a second. This is quite normal.

There is no **Undo** option, and no additional verification before the program saving, and the previous contents of the addressed **Flash** portion will be overwritten unconditionally, so please take good care when selecting the target portion of the **Flash** memory.

DIR Mode: Button LOAD

Loading is performed similarly to saving, but in the inverse manner. Command for loading from the external computer or some other unit via **UART** is simply pressing **LOAD**, and when the data starts flowing and the internal program memory loading data, the same taskbar appears on the three vertical bars, but this time in the inverse direction, upside down, suggesting the data flow from the **I/O** connector to the unit. If the **LOAD** button is pressed unintentionally in **DIR** mode, the process can be stopped by pressing any key, and if the valid header is not received through the serial port, the contents of the internal program memory will not be destroyed.

Command for program loading from the internal Flash memory is **ALT-LOAD**, but before that the same selection of the **Flash** portion should be performed. Every time prior to the program loading from the internal Flash, automatic safety writing of the current **Program Memory** contents to the location **15** is performed. So if you loaded the program from the **Flash** memory unintentionally (for instance, if you intended to perform **Save**), the contents of the internal **Program Memory** is wiped out, but there is a spare copy in the location **15** and we can load the program from it. The only thing that can be really destructive, is if you perform unwanted loading twice, as the new safety writing, which is performed automatically at the **LOAD** command, could wipe out the **Flash** location **15**, this time without the spare copy.

It means that Flash location **15** has the special function and can not be used for program storage. Command to write to this location (**ALT-SAVE** with location **15** selected) will be ignored by the system, but loading from location **15** will be executed normally (this time without automatic spare copying).

Here are the rules for serial **Save/Load**:

- Serial Port settings: **9600, N, 8, 1** (not affected by **SFR** setting in **SerCtrl** register)
- Hardware considerations: **3V CMOS** Logic levels (**Tx/Rx** on **I/O connector** only)
- **Tx/Rx** on **SAO connector** cannot be used for program saving or loading
- Software protocol is same as for LOAD command

DIR Mode: Button CLOCK

When the **Clock** button is pressed in DIR mode, the result of the current operation and **Flags** are latched in the **Master** Flip-Flops of the **Accumulator** and the **Status** register, respectively (note that every instruction enables or disables latching for the Accumulator and every Flag separately, depending on the instruction context).

但是，如果没有任何东西要保存（如果整个程序存储器为空），则指示灯SAVE将闪烁，并且没有数据将发送到串行端口。在这种情况下，任何按钮都将停止闪烁，并将设备返回到先前的状态。

在程序保存过程中，三个垂直解码指示条上会显示“”。

如果程序必须保存到内部闪存，则必须键入ALT-SAVE（按住ALT，同时按SAVE）。但在此之前，应该执行几个步骤。首先，选择程序将被保存的闪存部分的地址。共有15个可用部分（第16个部分具有特殊功能），应通过按住ALT按钮选择所需部分，同时通过按操作数Y字段中的按钮选择部分。有用的功能是，闪光灯占用显示在显示矩阵，所以你只需要匹配垂直指示栏旁边的所需部分。当你完成了Flash部分的选择，只需按下保存，而你仍然按住ALT。

写入闪存的速度比写入串行端口快得多，因此此处不显示任何数据。当驱动装置的主处理器在编程期间停止程序执行时，显示矩阵刷新被禁止，所有LED（除SAVE外）在几分之一秒内关闭。这很正常。

没有Undo选项，在程序保存之前没有额外的验证，并且所寻址的Flash部分的先前内容将被无条件覆盖，因此在选择Flash存储器的目标部分时请小心。

加载模式：按钮加载

加载的执行方式与保存类似，但方式相反。从外部计算机或其他单元通过USB加载的命令只需按下LOAD，当数据开始流动并且内部程序存储器加载数据时，相同的USB会出现在三个垂直条上，但这次是相反的方向，上下颠倒，表明数据从I/O连接器流向单元。如果在串行模式下无意中按下LOAD按钮，则可以通过按下任何键来停止该过程，并且如果未通过串行端口接收到有效的报头，则内部程序存储器的内容不会被破坏。

从内部闪存加载程序的命令是ALT-LOAD，但在此之前，应执行闪存部分的相同选择。每次从内部闪存加载程序之前，执行当前程序存储器内容到位置15的自动安全写入。所以如果你无意中从闪存中加载了程序（例如，如果你打算执行保存），内部程序存储器的内容被擦除，但在位置15有一个备用副本，我们可以从它加载程序。唯一可能真正具有破坏性的是，如果你执行不必要的加载两次，作为新的安全写入，该操作在加载命令中自动执行，可能会擦除闪存位置15，这次没有备用副本。

这意味着Flash位置15具有特殊功能，不能用于程序存储。写入此位置的命令（选择位置15的ALT-SAVE）将被系统忽略，但从位置15加载将正常执行（这次没有自动备份复制）。

以下是串行保存/加载的规则：

- 串行端口设置：9600、N、8、1（不受SerCtrl寄存器中SFR设置的影响）
- 硬件注意事项：3V CMOS逻辑电平（仅限I/O连接器上的Tx/Rx）-SAO连接器上的Tx/Rx不能用于程序保存或加载-软件协议与LOAD命令相同

时钟模式：按钮时钟

当时钟按钮按下处于复位模式时，当前操作的结果和标志将分别锁存到累加器的主触发器和状态寄存器中（请注意，每条指令分别启用或禁用累加器和每个标志的锁存，具体取决于指令上下文）。

Direct (DIR) Mode

Note: There are actually two Accumulators in DIR mode, one of them is **Register X** (available and displayed in the **Operand X** field) and the another one is **Register Y** (available and displayed in the **Operand Y** field). The one that is displayed is actually the destination of the operation. If there are two operands in the instruction, it will be **RX**, and if there is only one, it's **RY**. Switching of the two of them (which occurs when the Clock button is released and the next instruction does not use the same destination) can sometimes cause confusion.

After the **Clock** button is released, **Slave** Flip-flops in the **Accumulator** and the **Status** register perform the final latching and the result (if any) is safely stored in **Slave** Flip-flops. The new value of the destination register is written to **Rx** or **RY**, the new instruction is executed and the stage is ready for the next pressing of the **Clock** button.

直接（直接）模式

注意事项：实际上在累加器模式下有两个累加器，其中一个是寄存器X（可用并显示在操作数X字段中），另一个是寄存器Y（可用并显示在操作数Y字段中）。显示的那个实际上是操作的目的地。如果指令中有两个操作数，它将是RX，如果只有一个，它是RY。两者的切换（当时钟按钮被释放而下一个指令不使用相同的目的地时发生）有时会导致混淆。

时钟按钮释放后，累加器和状态寄存器中的从触发器执行最终锁存，结果（如果有）安全地存储在从触发器中。目的寄存器的新值写入Rx或RY，执行新指令，该阶段准备好下一次按下时钟按钮。

Single Step (SS) Mode

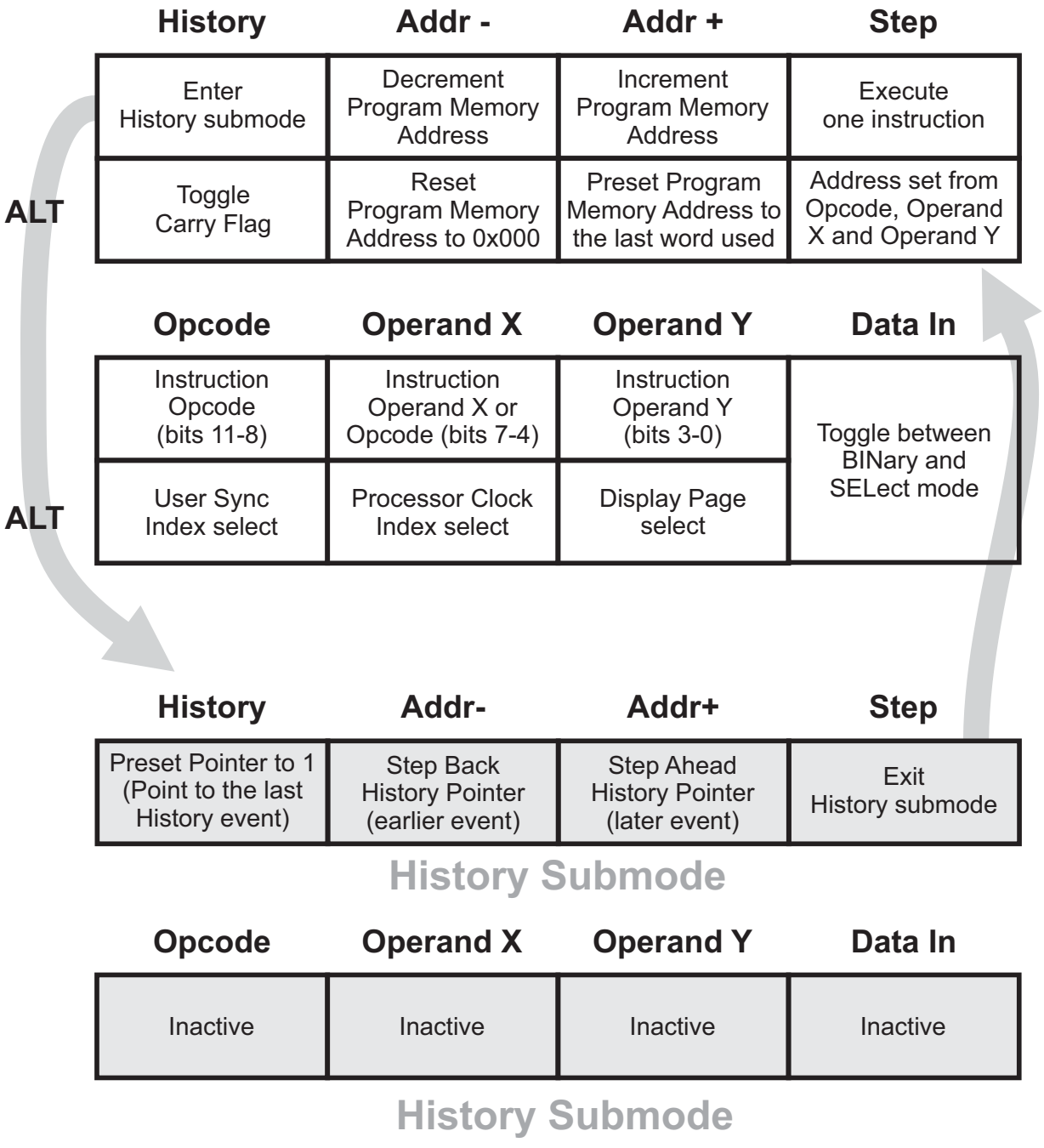
Single Step (SS) Mode

Single Step (SS) Mode supports manual stepping through the program written in the **Program Memory**. Modification of the contents of every program word is possible, but it is valid only once, for direct execution only (which is initiated by pressing "**STEP**"). Permanent modification of **Program Memory** is possible in **PGM** mode only.

Single Step (SS) Mode also contains the extra **HISTORY** submode, which allows reviewing of the last **127** steps performed in **Single Step (SS)** mode.

Display Matrix in **SS** mode shows the contents of **Data Memory** on the selected **Page**, but while the button **ALT** is pressed, **Page 0** (right) and **Page 1** (left) are unconditionally displayed.

Indicator **Carry** shows the **Carry Flag** state, but it cannot be modified manually in **SS** mode.



单步（SS）模式

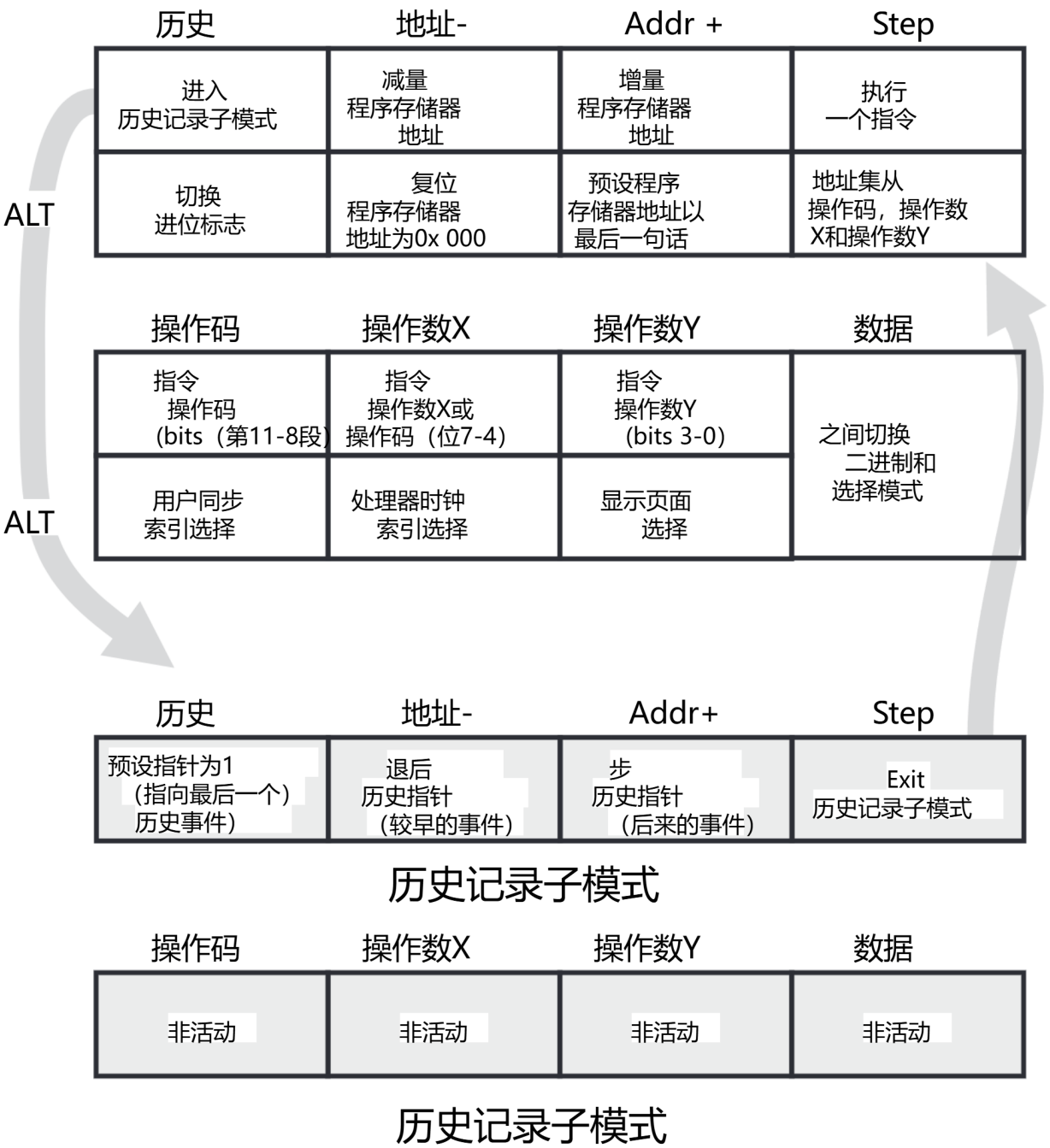
单步（SS）模式

单步（SS）模式支持手动单步执行写入程序存储器中的程序。可以修改每个程序字的内容，但仅对直接执行有效一次（按下“STEP”启动）。程序存储器的永久修改只能在PGM模式下进行。

单步（SS）模式还包含额外的历史子模式，允许查看单步（SS）模式中执行的最后127步。

SS模式下的显示矩阵显示所选页面上的数据存储器内容，但按下ALT按钮时，无条件显示页面0（右）和页面1（左）。

进位指示器显示进位标志状态，但在SS模式下不能手动修改。



Single Step (SS) Mode

Single Step (SS) Mode: Button HISTORY

Button **HISTORY** is used to enter **History** submode, which allows you to review the last **127** steps executed in **SS** mode. All visible indicators show the state of registers and logic levels when the event was executed and automatically recorded.

Data Memory contents of the page which was selected is also displayed, and when the button **ALT** is pressed in **History** submode, contents of Pages **0** and **1** (which were valid when the event was recorded) is also displayed.

Program Counter (PC) indicator shows the **Program Memory Address** of the recorded event, and when **ALT** is pressed, it displays the **History Pointer** state. That means that you can see how "deep" you are in **History** steps (how old is the displayed event, in executed steps).

Indicator **SS** blinks, signifying that the unit is in the **History** submode. Pressing **ADDR-** or **ADDR+** decrements or increments the **History Pointer**. When pressing of **ADDR-** or **ADDR+** causes the corresponding indicators to turn on, that means that pointer reached the start **(1)** or the end **(127)** of the History buffer.

The general rule in History mode is that you can see all program parameters and registers, but you can't modify anything.

Pressing the button **STEP** exits **HISTORY** submode and returns back to the normal **SS** mode.

If **ALT** is also depressed, then button **HISTORY** only toggles the **CARRY** flag.

Single Step (SS) Mode: Button ADDR -

Button **ADDR-** in **SS** mode decrements the 12-bit **Program Memory Address** pointer by 1.

If **ALT** is also depressed, then button **ADDR-** resets Program Memory Address to **0000 0000 0000**. Also, the whole **Data Memory**, **Stack Pointer** and **Page** register are cleared to zero.

If the current Program Memory Address is **0000 0000 0000**, decrementing will wrap the new value to **1111 1111 1111**, which is the last program word in the memory.

In History submode, button **ADDR-** steps back the History pointer (earlier event).

Single Step (SS) Mode: Button ADDR +

Button **ADDR+** in **SS** mode increments the 12-bit **Program Memory Address** pointer by 1. If **ALT** is depressed, then button **ADDR+** sets Program Memory Address to the last program word used (the last one which does not contain value **0000 0000 0000**).

If the current Program Memory Address is **1111 1111 1111**, which is the last program word in the memory, incrementing will wrap the new value to **0000 0000 0000**.

In History submode, button **ADDR+** steps ahead the History pointer (later event).

单步（SS）模式

单步（SS）模式：按钮历史

按钮HISTORY用于进入历史子模式，允许您查看SS模式下执行的最后127个步骤。所有可见指示器显示事件执行和自动记录时寄存器和逻辑电平的状态。

还将显示所选页面的数据存储器内容，当在历史子模式下按下ALT按钮时，还将显示页面0和1的内容（记录事件时有效）。

程序计数器（PC）指示器显示记录事件的程序存储器地址，当按下ALT时，它显示历史指针状态。这意味着您可以看到您在History步骤中的“深度”（在执行的步骤中，显示的事件有多老）。

指示灯SS闪烁，表示装置处于历史记录子模式。按ADDR-或ADDR+可递减或递增历史指针。当按下ADDR-或ADDR+导致相应的指示灯亮起时，这意味着指针到达了历史缓冲区的开始（1）或结束（127）。
历史模式的一般规则是，你可以看到所有的程序参数和寄存器，但你不能修改任何东西。

按下STEP按钮退出HISTORY子模式并返回正常SS模式。

如果ALT也被按下，则按钮HISTORY仅切换CARRY标志。

单步（SS）模式：按钮地址-

按钮ADDR-在SS模式下，12位程序存储器地址指针减1。

如果ALT也被按下，则按钮ADDR-将程序存储器地址重置为0000 0000 0000。此外，整个数据存储器、堆栈指针和页寄存器被清零。

如果当前程序存储器地址为0000 0000 0000，则递减将使新值回绕到1111 1111 1111，这是存储器中的最后一个程序字。

在历史记录子模式下，按钮ADDR-后退历史记录指针（较早的事件）。

单步（SS）模式：按钮ADDR +

SS模式下的按钮ADDR+使12位程序存储器地址指针递增1。如果按下ALT，则按钮ADDR+将程序存储器地址设置为使用的最后一个程序字（不包含值0000 0000 0000的最后一个程序字）。

如果当前程序存储器地址是1111 1111 1111，这是存储器中的最后一个程序字，则递增会将新值包装为0000 0000 0000。

在历史子模式下，按钮ADDR+步进历史指针（稍后事件）。

Single Step (SS) Mode

Single Step (SS) Mode: Button STEP

When the button **STEP** is pressed in SS mode, one program step (execution of the current instruction) is performed. When the button is depressed, the result of the current operation and **Flags** are latched in the **Master** Flip-Flops of the **Accumulator** and the **Status** register, respectively (note that every instruction enables or disables latching for the Accumulator and every Flag separately, depending on the instruction context).

After the **STEP** button is released, **Slave** Flip-flops in the **Accumulator** and the **Status** register perform the final latching and the result (if any) is safely stored in **Slave** Flip-flops. The new value of the destination register is, in most cases, written to the destination address (which depends on the instruction), then the **Program Counter** is incremented by **1** (or preset to the new value if the instruction caused program branching). Then the contents of the **Program Memory**, addressed by **Program Counter**, is read, latched and displayed by the indicators in **Opcode, Operand X** and **Operand Y**. The execution of the new instruction is simulated and all indicators show the correct state of the internal registers and data paths, but storing the result of the instruction has to wait for the new **STEP** cycle.

At this moment, when the system is waiting for the new **STEP** command, you can modify every bit in the instruction, and watch interactively how the processor registers react to the new values. The data which is supposed to be written at the destination, appears at the 4-bit **Accumulator** inputs, and the flags at the **Status Register** inputs (upper row). However, they will not be written to the destination if the instruction does not allow that. Some instructions don't write data to the destination (e.g. **CP Y** or **SKIP F, M**) and many instructions affect only some flags, if any. That's why some flags are shifted down in the register when the **STEP** button is pressed, and some are not. You can see which are affected by which instructions on the table at the back (bottom layer) of the unit PCB, or in the instruction list (PDF file "**INSTRUCTION SET**").

Every modification of the instruction is taken into account, and when the **STEP** button is pressed, the instruction in **Opcode, Operand X** and **Operand Y** registers will be executed, even if the **Program Memory** contains some other data. But the modification will be valid for only one execution, as the **Program Memory** contents were not modified. The only mode which has the right to affect the Program Memory contents is the **PGM** mode.

Prior to every instruction execution in **SS** mode, which happens when the button **STEP** is released, all machine states, flags, registers and two portions of the **Data Memory** (the currently displayed, and the portion of **Page 0** and **Page 1**) are loaded into the **127**-block **History Buffer**. The contents of this buffer will be used if the **History** submode is invoked.

When **ALT** is depressed, button **STEP** does not execute code in **Opcode, Operand X** and **Operand Y**, but loads the contents of these registers to the **Program Counter** (same as **Addr Set** command in **PGM** mode). It is a fast way to preset **PC** to some predetermined value.

单步（SS）模式

单步（SS）模式：按钮STEP

当在SS模式下按下STEP按钮时，执行一个程序步骤（执行当前指令）。当按下按钮时，当前操作的结果和标志分别锁存到累加器的主触发器和状态寄存器中（请注意，每条指令分别启用或禁用累加器和每个标志的锁存，具体取决于指令上下文）。

释放STEP按钮后，累加器和状态寄存器中的从触发器执行最终锁存，结果（如果有）安全地存储在从触发器中。在大多数情况下，目标寄存器的新值被写入目标地址（取决于指令），然后程序计数器递增1（如果指令导致程序分支，则预设为新值）。然后，由程序计数器寻址的程序存储器的内容被读取、锁存并由操作码、操作数X和操作数Y中的指示器显示。模拟新指令的执行，所有指示器显示内部寄存器和数据路径的正确状态，但存储指令的结果必须等待新的STEP周期。

此时，当系统正在等待新的STEP命令时，您可以修改指令中的每一位，并以交互方式观察处理器寄存器对新值的反应。应该在目的地写入的数据出现在4位累加器输入端，标志出现在状态寄存器输入端（上行）。但是，如果指令不允许，它们将不会写入目的地。有些指令不向目的地写入数据（例如CP Y或SKIP F，M），许多指令只影响某些标志（如果有的话）。这就是为什么当按下STEP按钮时，寄存器中的一些标志会下移，而另一些则不会。您可以在单元PCB背面（底层）的表格或指令列表（PDF文件“INDUSTRY SET”）中查看哪些指令受哪些指令的影响。

指令的每一次修改都被考虑在内，当按下STEP按钮时，操作码、操作数X和操作数Y寄存器中的指令将被执行，即使程序存储器包含一些其他数据。但是修改将仅对一次执行有效，因为程序存储器内容未被修改。唯一有权影响程序存储器内容的模式是PGM模式。

在SS模式下的每个指令执行之前，当释放STEP按钮时，所有机器状态、标志、寄存器和数据存储器的两个部分（当前显示的部分以及页面0和页面1的部分）都被加载到127块历史缓冲区中。如果调用历史记录子模式，则将使用此缓冲区的内容。

按下ALT时，STEP按钮不执行操作码、操作数X和操作数Y中的代码，但将这些寄存器的内容加载到程序计数器（与PGM模式下的地址设置命令相同）。这是一个快速的方式来预置PC到一些预定值。

RUN Mode

RUN Mode

RUN Mode is used to control the execution of the program written in **Program Memory**. When the **RUN** button is pressed in **RUN** mode, most registers and pointers (**Stack, Page, Clock, Sync, WrFlags**) are cleared, and some of them are preset to default values (**Dimmer** to maximum brightness, and **SerCtrl** to **0011** binary, so the **Baud Rate** is set to **9600**). The whole **Data Memory** is cleared, and then the execution starts from the **Program Memory Address 0000 0000 0000**. So the program is always executed with the known initial values, and the further state of all control bits and variables is at the user's program responsibility.

After the program is terminated (which will happen if the processor detected the **Stack Error**, or when button **BREAK** is pressed), the same register clearing process is performed again.

Buttons in **Opcode / Operand X / Operand Y** fields are inactive in **RUN** mode, so the code can not be affected. However, the same buttons are active with **ALT** pressed, so it is possible to modify **Sync, Clock** and **Page** parameters. If the program is running, it has to be paused first to perform the modification of **Sync, Clock** and **Page**.

Note that **Sync, Clock** and **Page** are regular **SFR (Special Function Registers)** and that they can be preset at any time under the program control.

Fast	Pause	Break	Run
On/Off Toggle 10× Faster Clock and Sync (if possible)	Program Execution Pause / Resume	Terminate Program Execution	RUN Program From Program Memory

Active only at runtime

	Opcode	Operand X	Operand Y	Data In
	Inactive	Inactive	Inactive	Toggle between BINary and SELect mode
ALT	User Sync Index select	Processor Clock Index select	Display Page select	

RUN Mode: Button FAST

Button **FAST** is used for switching between normal and fast (**10×**) execution mode. When the **FAST** mode is activated, indicator **RUN** in **MODE** field blinks.

In **FAST** mode, indexes in **SYNC** and **CLOCK** variables are temporarily modified to point to the higher speed. If normal settings are already at the maximum speed or close to the maximum, **FAST** mode will have no or little effect.

Pressing the button **FAST** has no effect if the program is not running.

运行模式

运行模式

运行模式用于控制写入程序存储器的程序的执行。当在RUN模式下按下RUN按钮时，大多数寄存器和指针（堆栈、页面、时钟、同步、时钟标志）被清除，其中一些被预设为默认值（调光器设置为最大亮度，SerCtrl设置为0011二进制，因此波特率设置为9600）。清除整个数据存储器，然后从程序存储器地址0000 0000 0000开始执行。因此，程序总是以已知的初始值执行，并且所有控制位和变量的进一步状态由用户的程序负责。

程序终止后（如果处理器检测到堆栈错误，或按下BREAK按钮，则会发生这种情况），再次执行相同的寄存器清除过程。

在RUN模式下，操作码/操作数X /操作数Y字段中的“删除”无效，因此代码不会受到影响。但是，按下ALT时，相同的按钮处于活动状态，因此可以修改Sync、Clock和Page参数。如果程序正在运行，则必须首先暂停以执行同步，时钟和页面的修改。

请注意，同步，时钟和页面是常规的SFR（特殊功能寄存器），它们可以在程序控制下随时预置。

	Fast	暂停	打破	Run
	开/关开关10× 更快的时钟和 同步（如果可能）	程序执行 暂停/恢复	终止 程序 执行	运行程序 From 程序存储器

仅在运行时活动

ALT	操作码	操作数X	操作数Y	数据
	非活动	非活动	非活动	之间切换 二进制和 选择模式
	用户同步 索引选择	处理器时钟 索引选择	显示页面 选择	

运行模式：快速按钮

按钮FAST用于在正常和快速（10倍）执行模式之间切换。当快速模式被激活时，指示灯RUN in MODE（模式运行）字段闪烁。

在FAST模式下，SYNC和CLOCK变量中的索引被临时修改以指向更高的速度。如果正常设置已经处于最大速度或接近最大速度，则FAST模式将没有影响或影响很小。

如果程序未运行，则按下按钮FAST无效。

RUN Mode

RUN Mode: Button PAUSE

Button **PAUSE** stops execution temporarily. While the **PAUSE** mode is active, indicator **PAUSE** blinks. In **Pause** mode, it is possible to modify variables **Sync, Clock** and **Page** by holding button **ALT** and pressing buttons in the **Opcode, Operand X** and **Operand Y**. There is a text under these buttons to helps orientation and serves as the command reminder.

Exit from **PAUSE** is performed by pressing the button **RUN**. Program execution will be performed normally. Pressing the button **BREAK** in Pause mode quits the program execution.

Pressing the button **PAUSE** has no effect if the program is not running.

RUN Mode: Button BREAK

The only way to quit the program execution regularly is to press the button **BREAK**. It is also possible to stop the PAUSE mode and to quit the program execution with the button BREAK.

When the **BREAK** button is pressed in **RUN** mode, most registers and pointers (**Stack, Page, Clock, Sync, WrFlags**) are cleared, and some of them are preset to default values (**Dimmer** to maximum brightness, and **SerCtrl** to **0011** binary, so the **Baud Rate** is set to **9600**). The whole **Data Memory** is cleared, and the unit is ready to restart the same or load the new program, or to change the mode.

Pressing the button **BREAK** in RUN mode has no effect if the program is not running.

RUN Mode: Button RUN

Button **RUN** in **RUN** mode starts the program execution from the **Program Memory**, starting from the address **0000 0000 0000**.

When the **RUN** button is pressed, most registers and pointers (**Stack, Page, Clock, Sync, WrFlags**) are cleared, and some of them are preset to default values (**Dimmer** to maximum brightness, and **SerCtrl** to **0011** binary, so the default **Baud Rate** is set to **9600**). The whole **Data Memory** is cleared. So the program is always executed with the known initial values, and it can adjust them dynamically, as needed.

运行模式

运行模式：暂停按钮

按钮PAUSE暂时停止执行。PAUSE（暂停）模式激活时，PAUSE（暂停）指示灯闪烁。在同步模式下，可以通过按住ALT按钮并按下操作码、操作数X和操作数Y中的按钮来修改Sync、Clock和Page变量。在这些按钮下面有一个文本来帮助定位，并作为命令提醒。

通过按下按钮RUN（运行）退出PAUSE（暂停）。程序将正常执行。在中断模式下按下BREAK按钮退出程序执行。

如果程序未运行，则按下暂停按钮无效。

运行模式：按钮断开

定期退出程序执行的唯一方法是按下BREAK按钮。也可以停止暂停模式，并使用BREAK按钮退出程序执行。

当在RUN模式下按下BREAK按钮时，大多数寄存器和指针（堆栈、页面、时钟、同步、时钟标志）被清除，其中一些被预设为默认值（调光器设置为最大亮度，SerCtrl设置为0011二进制，因此波特率设置为9600）。整个数据存储器被清除，并且该单元准备重新启动相同的或加载新程序，或改变模式。

如果程序未运行，则在运行模式下按下BREAK按钮无效。

运行模式：按钮运行

RUN模式下的RUN按钮启动程序存储器中的程序执行，从地址0000 0000 0000开始。

当按下RUN按钮时，大多数寄存器和指针（堆栈、页面、时钟、同步、时钟标志）被清除，其中一些被预设为默认值（调光器设置为最大亮度，SerCtrl设置为0011二进制，因此默认波特率设置为9600）。整个数据存储器被清除。因此，程序总是使用已知的初始值执行，并且可以根据需要动态调整它们。

Program (PGM) Mode

Program (PGM) Mode

Except for the **LOAD** command, **Program (PGM) Mode** is the only mode which allows modification and writing to the **Program Memory**. The regular procedure is to set the **12-bit Program Word** (preset and displayed in the **Opcode, Operand X** and **Operand Y** fields), and then, assuming that the **Program Counter (PC)** points to the desired address, press button **DEP+** (**Deposit** with post-increment). At that moment, the **12-bit Program Word** will be written to the internal **Program Memory** and the **Program Counter (PC)** will be incremented by **1**.

	Opcode	Operand X	Operand Y	Data In
	Instruction Opcode (bits 11-8)	Instruction Operand X or Opcode (bits 7-4)	Instruction Operand Y (bits 3-0)	Toggle between BINary and SELect mode
ALT→	User Sync Index select	Processor Clock Index select	Display Page select	

	Addr Set	Addr-	Addr+	Dep+
	Address set from Opcode, Operand X and Operand Y	Decrement Program Memory Address	Increment Program Memory Address	Write the current word, overwriting the old one, increment PC
ALT→	Delete the current word (move all subsequent words down)	Reset the Page and Program Memory Address to 0x000	Preset Program Memory Address to the last word used	Write the current word, moving (pushing) all subsequent words up
ALT / Both buttons→		Clear all Memory,**Data Memory** and most registers (Note: **NO UNDO**!)		

Program (PGM) Mode: ADDR SET

When the button **ADDR SET** is pressed in **PGM** mode, the contents of **Opcode, Operand X** and **Operand Y** registers are copied to the **Program Counter (PC)**. This is a convenient way to preset the desired **Program Address**. Note that **Opcode, Operand X** and **Operand Y** fields are not used for **Program Data**, like in all other cases, but for **Program Address**.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is transferred to the **Opcode, Operand X** and **Operand Y** fields.

程序（PGM）模式

程序（PGM）模式

除了LOAD命令，程序（PGM）模式是唯一允许修改和写入程序存储器的模式。常规程序是设置12位程序字（预设并显示在操作码、操作数X和操作数Y字段中），然后，假设程序计数器（PC）指向所需地址，按下按钮DEP+（带有后增量的存款）。此时，12位程序字将写入内部程序存储器，程序计数器（PC）将递增1。

	操作码	操作数X	操作数Y	数据
	指令 操作码 (bits（第11-8段)	指令 操作数X或 操作码（位7-4）	指令 操作数Y (bits 3-0)	之间切换 二进制和 选择模式
ALT	用户同步 索引选择	处理器时钟 索引选择	显示页面 选择	

	地址集	地址-	Addr+	Dep+
	地址集从 操作码，操作数 X和操作数Y	减量 程序存储器 地址	增量 程序存储器 地址	写入当前单词，删除旧单词，增加PC
ALT	删除当前单词（将所有后续单词下移）	重置页面并 程序存储器 地址为0x 000	预设程序 存储器地址以 最后一句话	写下当前单词，向上移动（推动）所有后续单词
ALT /两个按钮		清除所有内存、数据内存和 大多数寄存器（注意：NO UNDO！）		

编程（PGM）模式：地址设置

当在PGM模式下按下ADDR SET按钮时，操作码、操作数X和操作数Y寄存器的内容被复制到程序计数器（PC）。这是预设所需程序地址的方便方法。请注意，与所有其他情况一样，操作码、操作数X和操作数Y字段不用于程序数据，而是用于程序地址。

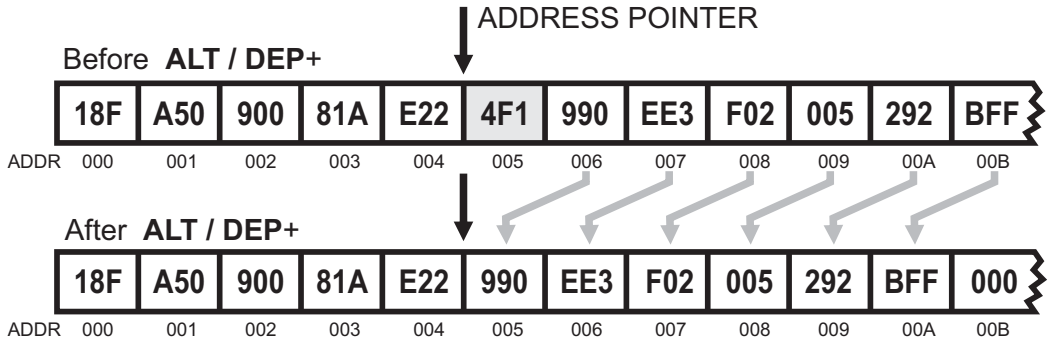
请注意，每次修改程序地址时，寻址内存位置的内容将传输到操作码、操作数X和操作数Y字段。

Program (PGM) Mode

Program (PGM) Mode: ALT / ADDR SET

If **ALT** is depressed, command **ADDR SET** deletes the current Program Word and moves all the subsequent words in the Program Memory one place down, thus overwriting the current memory location. The **Program Address Pointer** is unchanged. There is no **UNDO**.

Here is an example:



Note: Command **ALT / ADDR SET** moves all subsequent program words one position down, which typically means that all symbol names should be modified. This has no effect only if the current instruction pointer is near the end of the written program and there are no symbol names behind it.

Program (PGM) Mode: ADDR-

Button **ADDR-** in **PGM** mode decrements the 12-bit **Program Memory Address** pointer by 1.

If the current Program Memory Address is **0000 0000 0000**, decrementing will wrap the new value to **1111 1111 1111**.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is automatically transferred to the **Opcode, Operand X** and **Operand Y** fields.

Program (PGM) Mode: ALT / ADDR-

If **ALT** is depressed, then button **ADDR-** resets Program Memory Address to **0000 0000 0000**. Also, the whole **Data Memory**, **Stack Pointer** and **Page** register are cleared to zero.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is automatically transferred to the **Opcode, Operand X** and **Operand Y** fields

Program (PGM) Mode: ADDR+

Button **ADDR+** in **PGM** mode increments the 12-bit **Program Memory Address** pointer by 1..

If the current Program Memory Address is **1111 1111 1111**, which is the last program word in memory, incrementing will wrap the new value to **0000 0000 0000**.

Note that every time when the **Program Address** is modified, the contents of the addressed memory location is automatically transferred to the **Opcode, Operand X** and **Operand Y** fields.

Program (PGM) Mode: ALT / ADDR+

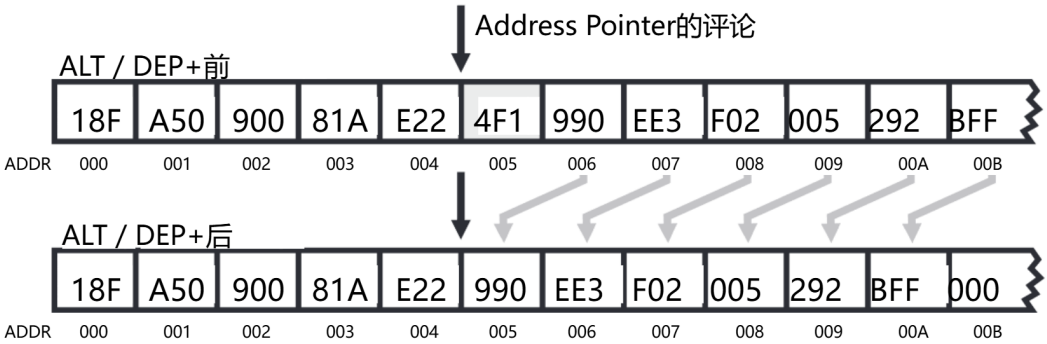
If **ALT** is depressed, command **ADDR+** sets the Program Address Pointer (current program word) to the last word used in program (the last word which is not 0x000).

程序（PGM）模式

Program（PGM）模式：ALT/ADDR SET

如果ALT被按下，命令ADDR SET删除当前程序字，并将程序存储器中的所有后续字向下移动一个位置，从而删除当前存储器位置。程序地址指针不变。没有撤销。

下面是一个示例：



注意：命令ALT / ADDR SET将所有后续程序字向下移动一个位置，这通常意味着所有符号名称都应被修改。只有当当前指令指针接近所写程序的结尾并且后面没有符号名称时，这才没有效果。

编程（PGM）模式：ADDR-

按钮ADDR-在PGM模式下，12位程序存储器地址指针减1。

如果当前程序存储器地址为0000 0000 0000，则递减将使新值回绕为1111 1111 1111。

请注意，每次修改程序地址时，寻址内存位置的内容会自动传输到操作码、操作数X和操作数Y字段。

程序（PGM）模式：ALT / ADDR-

如果按下ALT，则按钮ADDR-将程序存储器地址重置为0000 0000 0000。
此外，整个数据存储器、堆栈指针和页寄存器被清零。

请注意，每次修改程序地址时，寻址内存位置的内容会自动传输到操作码、操作数X和操作数Y字段

编程（PGM）模式：ADDR+

PGM模式下的按钮ADDR+使12位程序存储器地址指针递增1。

如果当前程序存储器地址是1111 1111 1111，这是存储器中的最后一个程序字，则递增将使新值返回到0000 0000 0000。

请注意，每次修改程序地址时，寻址内存位置的内容会自动传输到操作码、操作数X和操作数Y字段。

编程（PGM）模式：ALT / ADDR+

如果按下ALT，命令ADDR+将程序地址指针（当前程序字）设置为程序中使用的最后一个字（不是0x000的最后一个字）。

Program (PGM) Mode

Program (PGM) Mode: ALT / ADDR- / ADDR+

If **ALT** is depressed, pressing **ADDR-** and **ADDR-** simulteneously clears the whole **Program Memory, Data Memory** and most registers. There is no **UNDO**.

Program (PGM) Mode: DEP+

Button **DEP+** in **PGM** mode is used to store the contents of the **Opcode, Operand X** and **Operand Y** fields to the program memory, at the location defined by the **Program Memory Address** , overwriting the previous contents (no **UNDO**). After storing the **Program Word** to the **Program Memory,** the **Program Counter (PC)** is automatically incremented by **1**. Then the contents of the new location (addressed by the new value of **PC**) is automatically transferred to the **Opcode, Operand X** and **Operand Y** indicators.

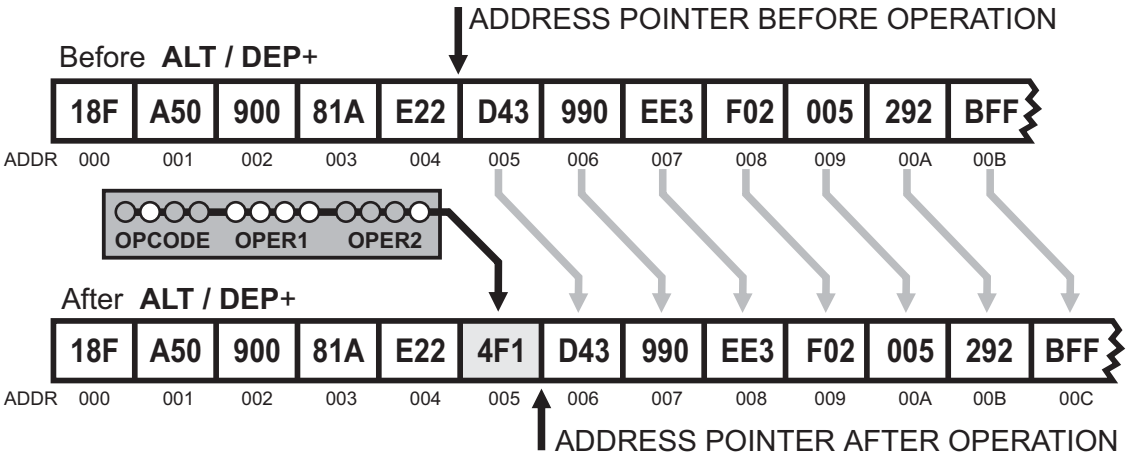
This means two things, which should be kept in mind during program writing. First, if the **DEP+** button is repeatedly pressed in **PGM** mode without modifying the **Program Word** contents, it will not affect the contents of the **Program Memory**, as every location is first read and stored to the **Opcode, Operand X** and **Operand Y** registers, and then rewritten to the memory unchanged. And second, the contents of the **Opcode, Operand X** and **Operand Y** registers are not written to the **Program Memory** until the **DEP+** button is pressed. Especially if only one word has to be modified, it is easy to make the mistake and switch to **PGM** mode, then find and modify the **Program Word**, and then hurry back to **RUN** mode to test it. Program memory is not modified if **DEP+** was not pressed after the modification was performed in the **Opcode, Operand X** and **Operand Y** fields.

If you use the analogy with Text Editor, **DEP+** command is similar to writing text in Overwrite mode, and **ALT / DEP+** in pushwrite mode.

Program (PGM) Mode: ALT / DEP+

If **ALT** is depressed, command **DEP +** not only writes the new word to the Program Memory, but also moves all the subsequent words in the Program Memory one place up, thus freeing one memory location. Then it duplicates the current word on the new location. There is no **UNDO**.

Here is the example:



Note: Command **ALT / DEP+** moves all subsequent program words one position up, which typically means that all symbol names should be modified. This has no effect only if the current instruction pointer is near the end of the written program and there are no symbol names behind it.

If you use the analogy with Text Editor, **ALT / DEP+** command is similar to writing text in Pushwrite mode, and **DEP+** in Overwrite mode.

程序（PGM）模式

项目模式：ALT/ADDR/ADDR +

如果按下ALT，按ADDR-和ADDR-同时清除整个程序存储器、数据存储器和大多数寄存器。没有撤销。

程序（PGM）模式：DEP+

PGM模式下的按钮DEP+用于将操作码、操作数X和操作数Y字段的内容存储到程序存储器中，位于程序存储器地址定义的位置，恢复先前的内容（无UNDO）。将程序字存储到程序存储器后，程序计数器（PC）自动递增1。然后，新位置的内容（由PC的新值寻址）自动传输到操作码、操作数X和操作数Y指示器。

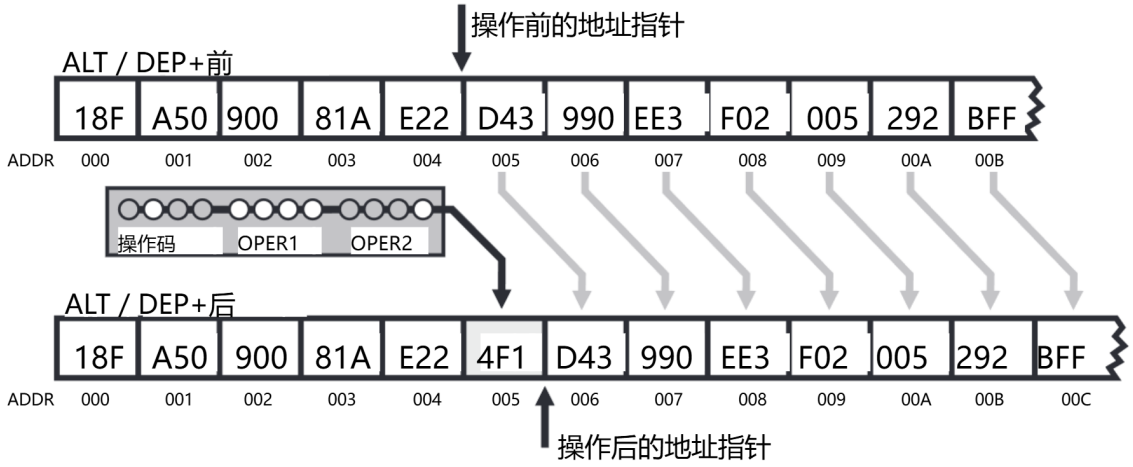
这意味着在编写程序时应该记住两件事。首先，如果在PGM模式下重复按下DEP+按钮而不修改程序字内容，则不会影响程序存储器的内容，因为每个位置首先被读取并存储到操作码、操作数X和操作数Y寄存器中，然后以不变的方式重写到存储器中。其次，操作码、操作数X和操作数Y寄存器的内容不会写入程序存储器，直到按下DEP+按钮。特别是如果只需要修改一个字，很容易出错并切换到PGM模式，然后找到并修改程序字，然后匆忙返回RUN模式进行测试。如果在操作码、操作数X和操作数Y字段中执行修改后没有按DEP+，则程序存储器不会被修改。

如果你使用文本编辑器的类比，DEP+命令类似于在覆盖模式下写文本，ALT / DEP+在推写模式下写文本。

程序（PGM）模式：ALT / DEP+

如果ALT被按下，命令DEP +不仅将新的字写入程序存储器，而且还将程序存储器中的所有后续字向上移动一个位置，从而释放一个存储器位置。然后它在新位置上复制当前单词。没有撤销。

下面是一个例子：



注意：命令ALT / DEP+将所有后续程序字上移一个位置，这通常意味着所有符号名称都应修改。只有当当前指令指针接近所写程序的末尾并且后面没有符号名时，这才没有效果。

如果你使用文本编辑器的类比，ALT / DEP+命令类似于在Pushwrite模式下写文本，DEP+在Overwrite模式下写文本。

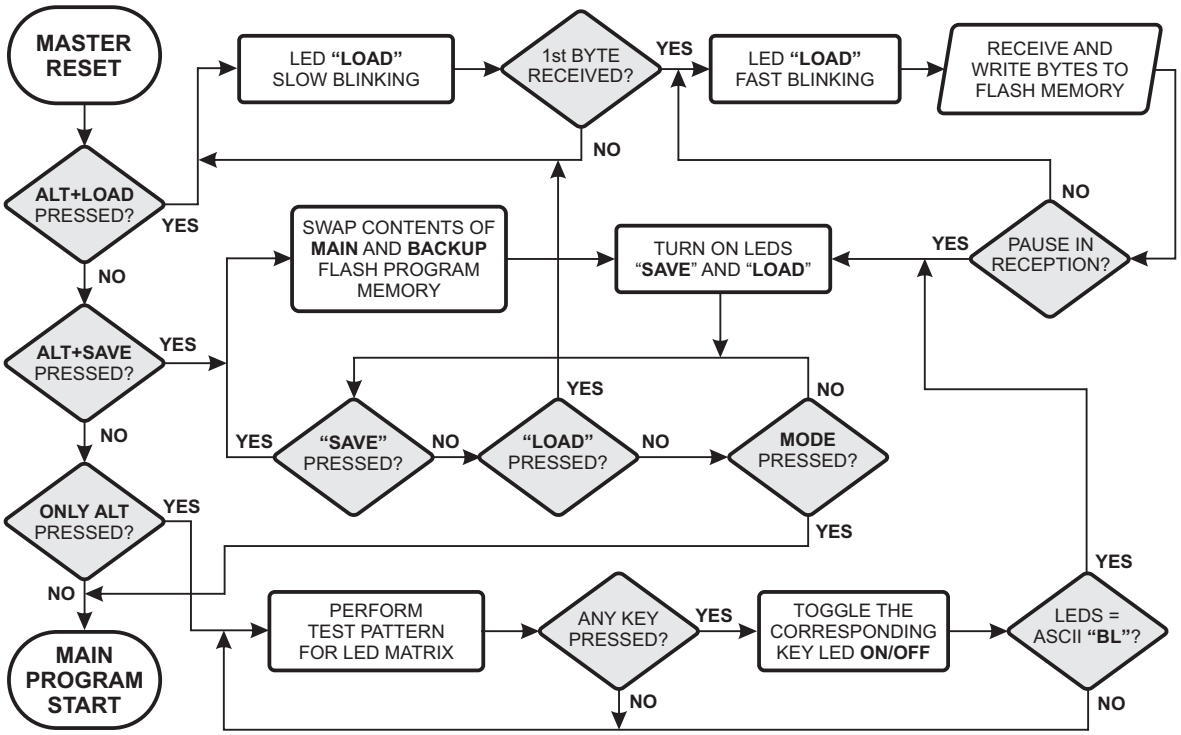
Special Modes

TEST MODE

Test mode is used for hardware testing. To enter **Test mode**, **ALT** button should be pressed at the moment when the unit starts operation after the hard **Reset**. There is no **Reset** button, but the **Reset** procedure may be performed by one of the two following methods:

- Disconnect and reconnect one of batteries, or
- Short pins **Ground** (third from the right) and **Reset** (the rightmost) on the I/O connector. Take care not to short pin **+3V** (which is between **Ground** and **Reset**) with **Ground**.

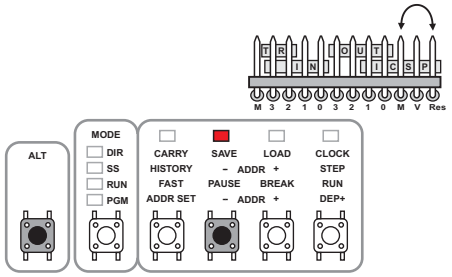
In **Test mode**, at the first moment all **LEDs** are in dynamic ON state, with the 1/3 duty cycle marching pattern. When the first button is pressed, all **LEDs** remain turned ON, except the display matrix and the lowest row, above buttons. Now you can toggle **On/Off** every **LED** individually, and thus test every button individually. When all the buttons are tested (all **LEDs** turned **ON**), the **TEST** mode is terminated and the unit returns to the normal operation.

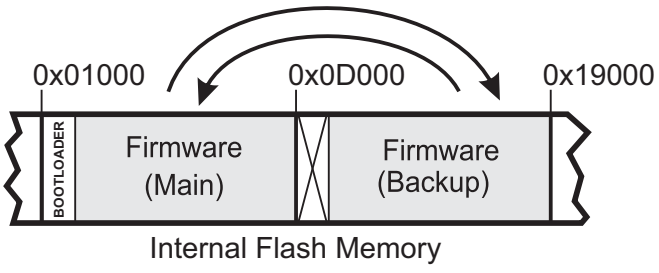


BOOTLOAD MODE

There are two functions in the **Bootload** mode: **SAVE**, which makes the backup copy of the current firmware in the extra space of internal Flash Memory, and **LOAD**, which downloads the new firmware from the computer, via **Serial Port**.

To make the backup copy of the current firmware, keep the buttons **ALT** and **SAVE** pressed during the **Reset** procedure (connecting the battery or shorting pins **Reset** and **Ground**). This swaps the **Main** and **Backup Program Memory** regions of the **MCU**, which takes about **5 sec**:





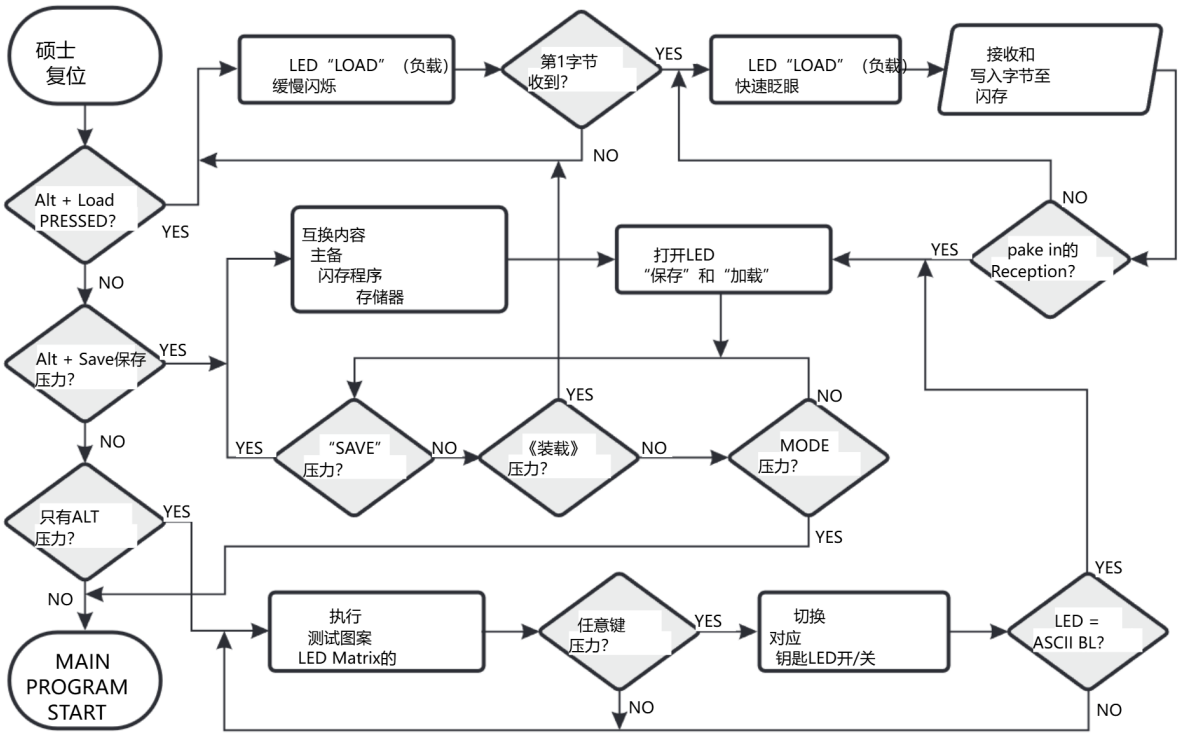
特殊模式

测试模式

测试模式用于硬件测试。要进入测试模式，应在硬复位后装置开始运行时按下ALT按钮。没有重置按钮，但可以通过以下两种方法之一来执行重置程序：

- 断开并重新连接其中一个电池，或-将I/O连接器上的接地（右数第三个）和复位（最右侧）引脚短接。
  注意不要将+3V引脚（接地和复位之间）与接地短路。

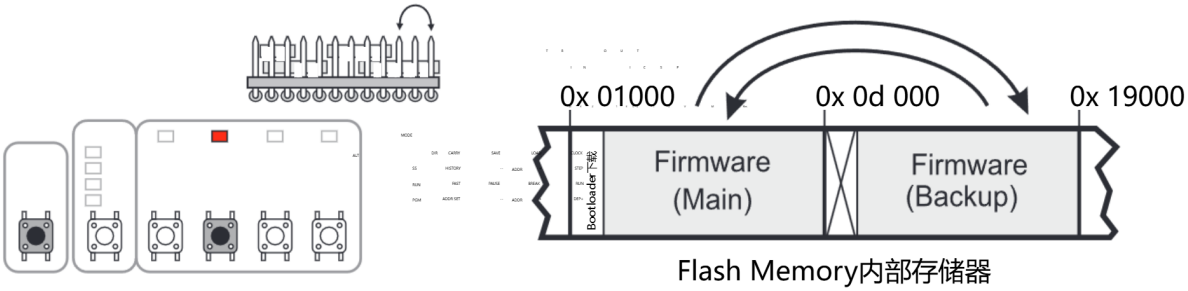
在测试模式下，在第一时刻，所有LED都处于动态ON状态，具有1/3占空比步进模式。当按下第一个按钮时，除显示矩阵和按钮上方的最低行外，所有LED均保持打开状态。现在，您可以单独打开/关闭每个LED，从而单独测试每个按钮。当所有按钮均已测试（所有LED均打开）时，测试模式终止，装置返回正常操作。



引导模式

引导加载模式有两个功能：保存，在内部闪存的额外空间中备份当前固件的副本，以及加载，通过串行端口从计算机下载新固件。

要备份当前固件，请在重置过程中（连接电池或短路引脚重置和接地）保持按下ALT和SAVE按钮。这将交换MCU的主程序存储器和备份程序存储器区域，大约需要5秒：

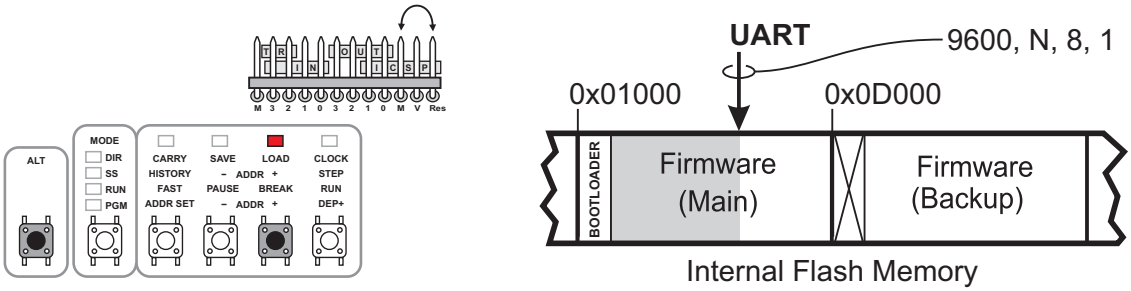


Flash Memory内部存储器

Special Modes

To download the new firmware immediatelly from the Serial Port, keep the buttons **ALT** and **LOAD** pressed during **Reset**. LED **LOAD** will blink slowly until the firmware starts loading, and than it will switch to fast blinking. This takes about **30 sec** or more, depending on the binary file length.

UART settings are **9600, N, 8, 1**. Settings in **SFR 0xF3 (SerCtrl)** do not affect the **Baud Rate** in **Bootload Mode** or for **Program Save/ Load**. Only pin **Rx** in the I/O connector is active, not the pin **Rx** on the **SAO** port. There is no transmitting from the unit, so although the pin **Tx** can be connected, it has no effect.



After any of the two procedures (**Swap** or **Bootload**), LEDs **SAVE** and **LOAD** are both turned **ON**, which means that the unit is still in **Bootload** mode. Only two buttons are active in this mode:

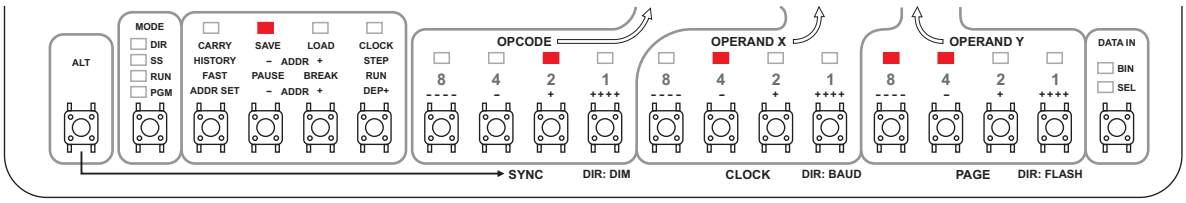
MODE: Return to normal mode
LOAD: Download the new firmware again from the **Serial Port**.

If there was no meaningful firmware in the **Backup** area and the **Program Memory Swap** is performed, command **MODE** has no sense, as the program will execute instructions from the unprogrammed area and will probably crash. In this case, **ALT-SAVE** with **RESET** should be performed again, to restore the old firmware, or **ALT-LOAD** during **RESET**, and the proper firmware downloading.

After the **bootload** process is complete, you can switch back to the normal mode, by pressing the button **MODE**. The firmware parameters **Version/Revision/Year/Month/Date** will be indicated at the top of the **LED** matrix, and the calculated **Checksum** at the bottom of the matrix. If the checksum matches the 16-bit number published together with the version, it means that the **bootload** process was **OK**. Between the **Version** and **Firmware Checksum** data (in rows **10** and **11**), there is the **Bootload Firmware Checksum**, in the same format as the **Main Firmware Checksum**. The **Bootload Firmware** area is write protected, so the checksum should not change in any circumstances.

Checksums are indicated in the **Big Endian** format, which means that the **High Byte** is displayed in the row **14 (0xE)** and the **Low Byte** in the row **15 (0xF)**. The Most Significant Bit (**MSB**) is at the left side. After the first button is pressed, **Version** and **Checksum** data will disappear from the LED matrix and the only way to see it again is to reset the unit again.

There is the alternate way to enter **Bootload** mode: while the unit is in Test mode (when the LEDs are blinking or all schematics LEDs are ON), you can test every button by switching the corresponding LEDs ON/OFF. You can use it to adjust **LEDs** in the **Command** and **Opcode** fields to display the **ASCII** uppercase **“B” (01000010)**, and also uppercase **“L” (01001100)** in the **Operand X** and **Operand Y** fields (**BL** is the abbreviation of **BootLoader**). Here is how it should look like:

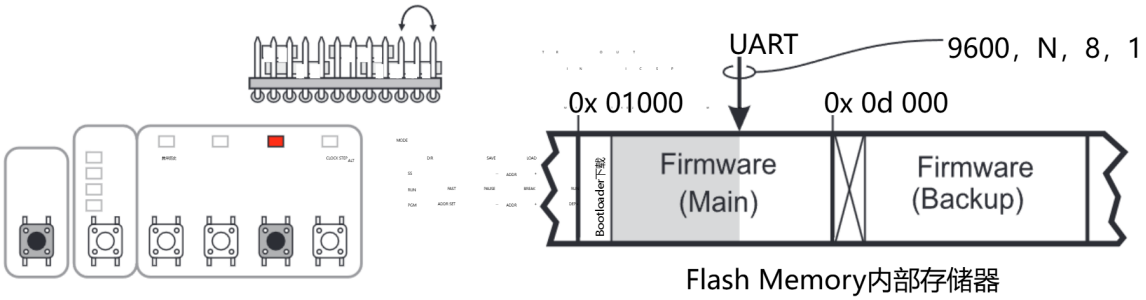


When the **BL** “passcode” is recognized, the unit is automatically switched to the **Bootload** mode. In this mode, LEDs **SAVE** and **LOAD** are turned ON, and only buttons **MODE, SAVE** and **LOAD** are active.

特殊模式

要立即从串行端口下载新固件，请在重置期间按下ALT和LOAD按钮。LED LOAD将缓慢闪烁，直到固件开始加载，然后它将切换到快速闪烁。这大约需要30秒或更长时间，具体取决于二进制文件的长度。

设置为9600、N、8、1。SFR 0xF 3（SerCtrl）中的设置不影响引导加载模式或程序保存/加载的波特率。只有I/O连接器中的Rx引脚有效，而SAO端口上的Rx引脚无效。没有从单位发射，所以虽然引脚Tx可以连接，它没有任何影响。



Flash Memory内部存储器

在两个程序（交换或引导加载）中的任何一个之后，LED SAVE和LOAD都打开，这意味着设备仍处于引导加载模式。在此模式下，只有两个按钮处于活动状态：

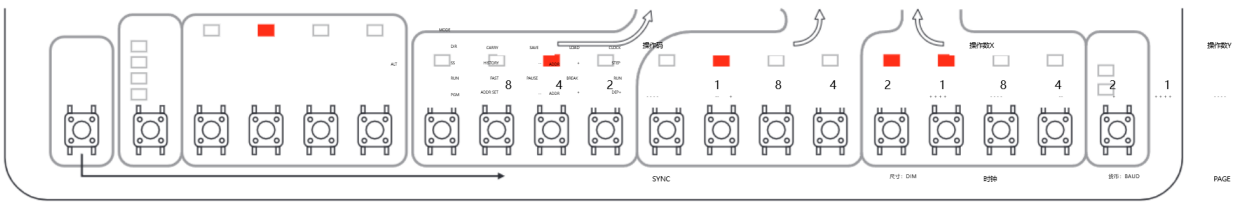
MODE：返回正常模式LOAD：从串口再次下载新固件。

如果备份区域中没有有意义的固件，并且执行了程序内存交换，则命令MODE没有意义，因为程序将执行来自未编程区域的指令，并且可能会崩溃。在这种情况下，应再次执行ALT-SAVE（使用固件升级），以恢复旧固件，或在固件升级期间执行ALT-LOAD（使用固件升级），并下载正确的固件。

引导加载过程完成后，您可以按MODE按钮切换回正常模式。固件参数版本/修订/年/月/日将显示在LED矩阵的顶部，计算的校验和显示在矩阵的底部。如果校验和与版本一起发布的16位数字匹配，则意味着引导加载过程正常。在版本和固件校验和数据之间（第10和11行），有引导加载固件校验和，格式与主固件校验和相同。引导加载固件区域是写保护的，因此校验和在任何情况下都不应该改变。

校验和以大端格式表示，这意味着高字节显示在第14行（0xE），低字节显示在第15行（0xF）。最高有效位（MSB）位于左侧。按下第一个按钮后，版本和校验和数据将从LED矩阵中消失，再次看到它的唯一方法是再次重置单元。

还有另一种进入引导加载模式的方法：当设备处于测试模式时（当LED闪烁或所有原理图LED亮起时），您可以通过打开/关闭相应的LED来测试每个按钮。您可以使用它来调整命令和操作码字段中的LED，以显示ASCII字符“B”（01000010），以及操作数X和操作数Y字段中的字符“L”（01001100）（BL是BootLoader的缩写）。它应该是这样的：



当BL“密码”被识别时，装置自动切换到引导加载模式。在此模式下，LED SAVE和LOAD打开，只有MODE、SAVE和LOAD按钮处于活动状态。

Special Modes

SYNTHETIC INSTRUCTIONS

Synthetic Instructions or **Pseudoinstructions** are instructions which are not present in the regular instruction set, so they have to be synthesized using existing (naitive) instructions. Some of sytnetic instructions need more than one naitive instruction to synthesize the new one.

Non-existing instruction		Replace with	
RLC RX, RY	Rotate Left **RY** through Carry	**MOV RX,RY** **ADDC RX, RY**	Note: Result is in **RX**
SL RX, RY	Shift Left **RY**	**MOV RX,RY** **ADD RX, RY**	Note: Result is in **RX**
LSR RY	Logical Shift Right **RY**	**AND R0, 0** **RRC RY**	
CPL R0	Complement **R0**	**XOR R0, 0xF**	
CPL RX, RY	Complement **RY**, write to **RX**	**MOV RX, 0xF** **SUB RX, RY**	
NEG RX, RY	Negate **RY**, write to **RX**	**MOV RX, 0** **SUB RX, RY**	
NOP	No Operation	**MOV R0, R0**	Note: Any Register except **R12** and **R13**

Although Synthetic Instructions are generally the good replacement for some non-existing instructions, care should be taken about how they affect the flags.

AUTOMATIC OFF

There is the internal countdown timer which counts independently of all other registers and, when it reaches zero, it turns the unit off, in the same way as the button command **OFF** should do. The upper nibble of the timer is in the **SFR** area, at the address **0xF9**, with the symbol name **AutoOff**.

Register **AutoOff** counts in the **10** minute resolution. It is readable and writeable, so when the processor writes a number in the **0-15** range, the unit will be turned of automatically in **10×N** minutes. Writing **0** to the register switches the unit momentarilly, and writing **15** will switch it off in **150** minutes (**2.5** hours).

At the Master Reset and **ON** command (button **ON-OFF** pressed while the unit is **OFF**), this register is preset to **2** (binary **0010**). That means that the unit will be switched **OFF** after **20** minutes if no other button is pressed in the meantime. At every button press (while the unit is **ON**), value **15** (binary **1111**) is written to **AutoOff** register, so it will be switched **OFF** after **2.5** hours of total inactivity.

Sometimes it is required that the program works longer time without human supervision or intervention. The simple way to solve this problem is to put the instruction which writes **1111** or some other value in the register **AutoOff**, in the main program loop.

特殊模式

合成指令

合成指令或伪指令是在常规指令集中不存在的指令，因此它们必须使用现有的（本机）指令合成。有些合成指令需要多条本机指令来合成新的指令。

不存在的指令		替换为	
RLC RX，RY	通过进位向左旋转RY	MOV RX，RY ADDC RX，RY	注：结果以RX表示
SL RX、RY	左移RY	MOV RX，RY ADD RX、RY	注：结果以RX表示
LSR RY	逻辑右移RY	AND R0，0 RRC RY	
CPL R0	补体R0	XOR R0，0xF	
CPL RX，RY	补码RY，写入RX	MOV RX，0xF SUB RX、RY	
NEG RX，RY	否定RY，写入RX	MOV RX，0 SUB RX、RY	
NOP	无操作	MOV R0，R0	注：任何登记册 R12和R13除外

虽然合成指令通常是一些不存在的指令的良好替代品，但应注意它们如何影响标志。

自动关断

有一个内部倒计时定时器，它独立于所有其他寄存器进行计数，当它达到零时，它会关闭该单元，与按钮命令OFF应该做的方式相同。定时器的上半字节位于SFR区域，地址为0xF9，符号名为AutoOff。

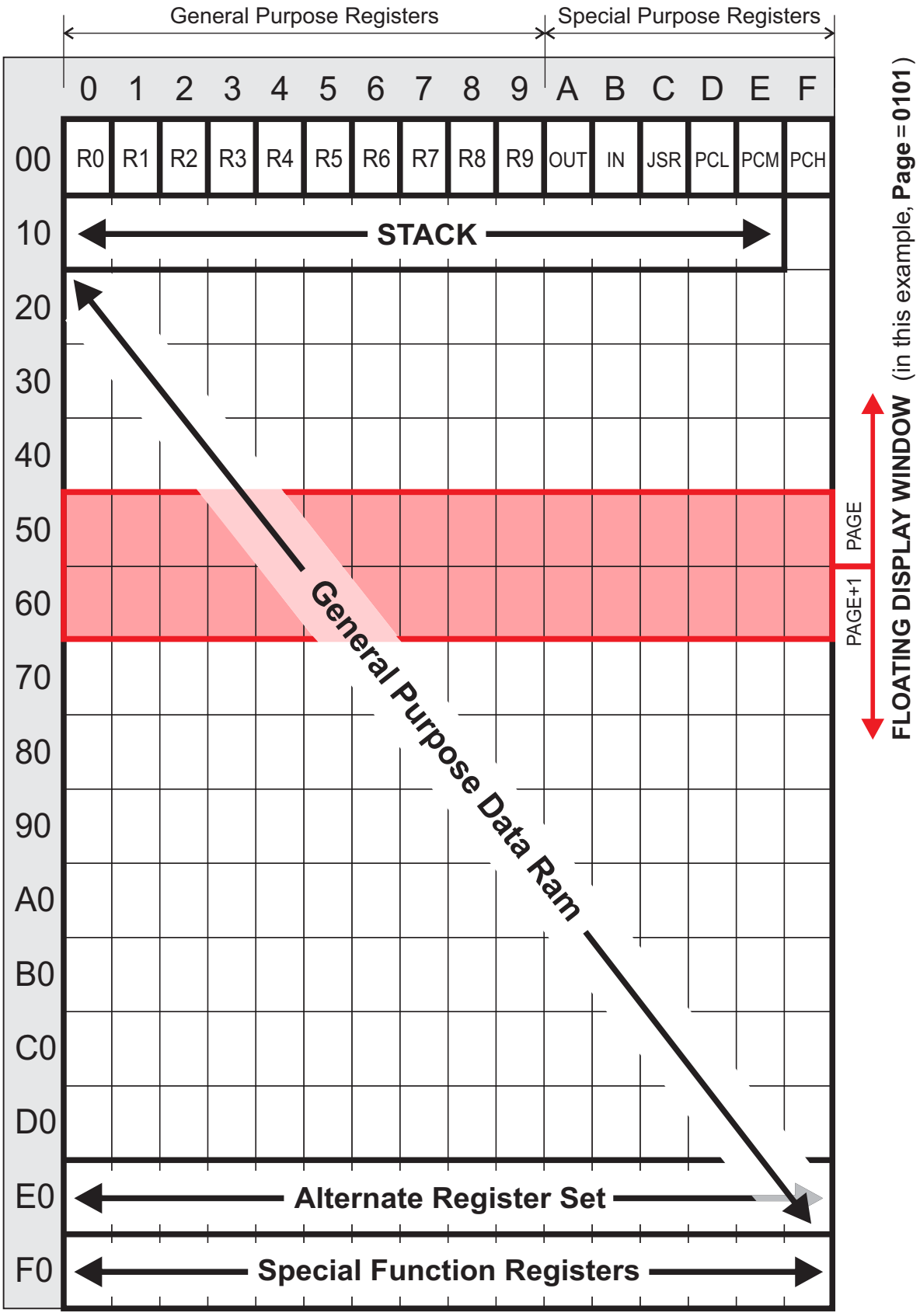
在10分钟分辨率中记录自动关闭计数。它是可读和可写的，所以当处理器写入0-15范围内的数字时，该单元将在10×N分钟内自动关闭。将0写入寄存器会立即切换该单元，而写入15则会将其关闭，

150 分钟（2.5小时）。

在主机复位和ON命令下（在装置关闭时按下ON-OFF按钮），该寄存器预设为2（二进制0010）。这意味着，如果在此期间没有按下其他按钮，则该装置将在20分钟后关闭。每次按下按钮时（设备处于打开状态时），值15（二进制1111）将写入AutoOff寄存器，因此在2.5小时的总不活动时间后，它将被关闭。

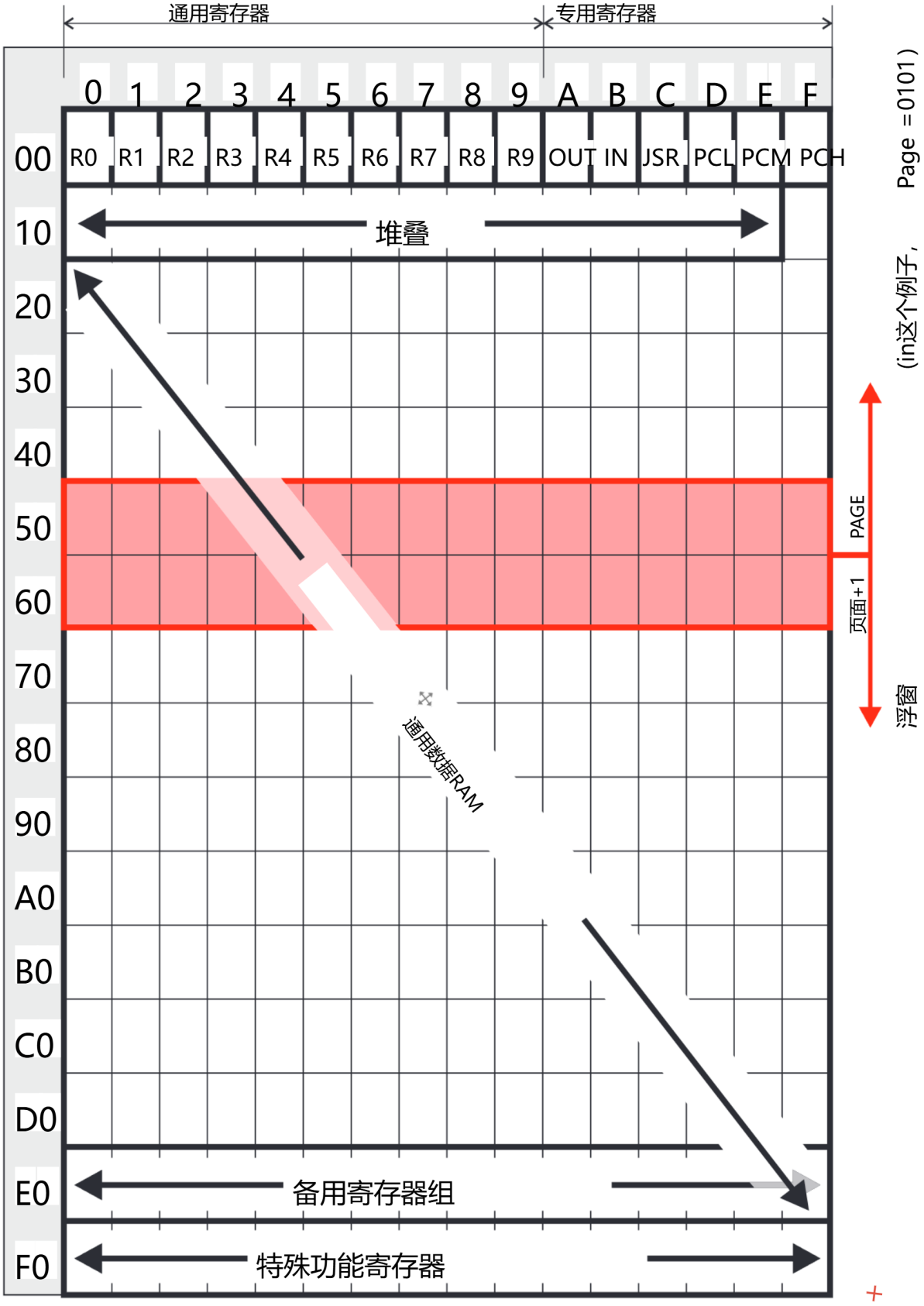
有时需要程序在没有人类监督或干预的情况下工作更长时间。解决此问题的简单方法是将写入1111或寄存器AutoOff中的其他值的指令放入主程序循环中。

Data Memory Organization (simulated processor)



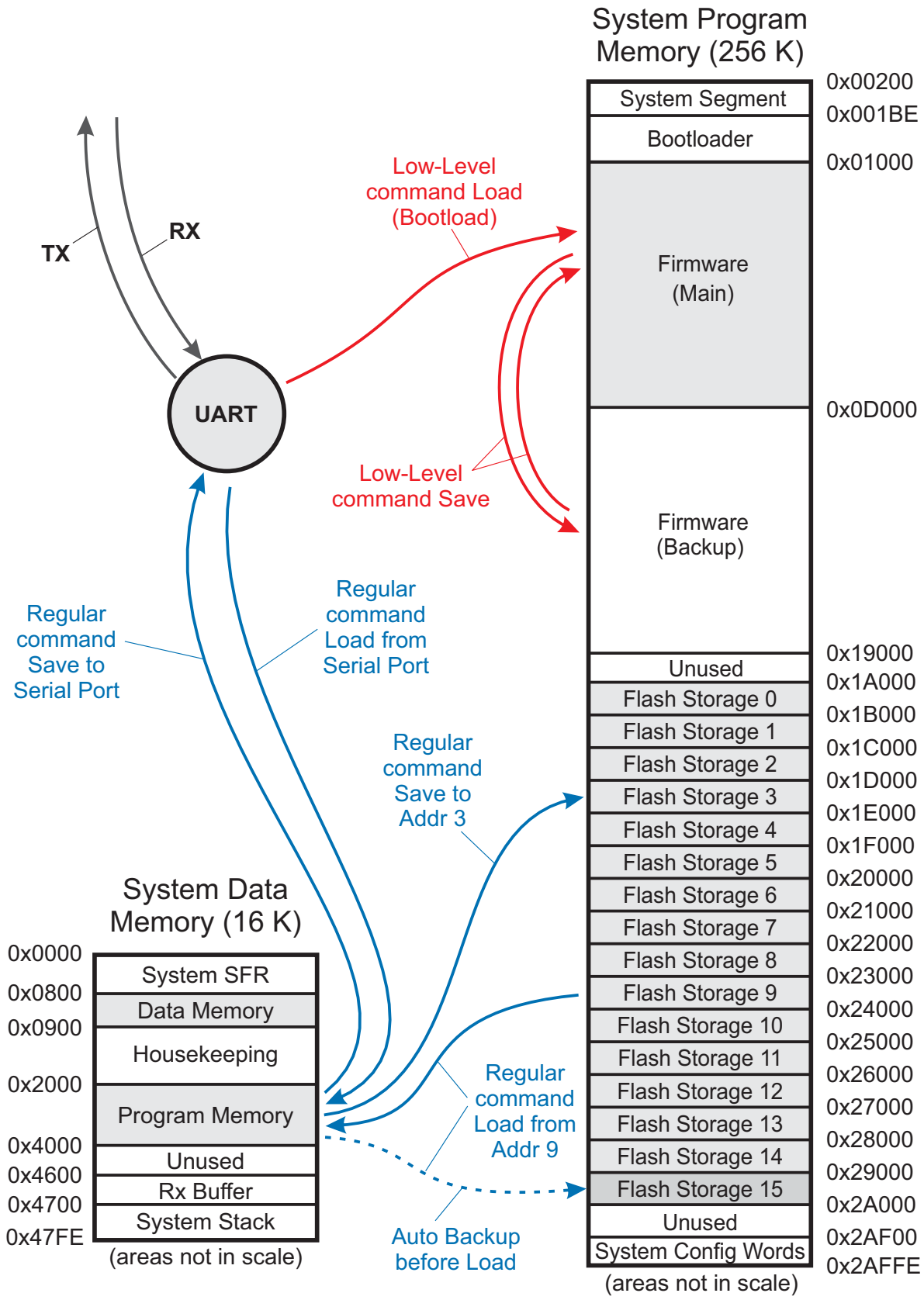
Note: this is the representation of the Data Memory for the hypothetical simulated processor, not the system processor PIC24FJ256GA704.

数据存储器组织（模拟处理器）



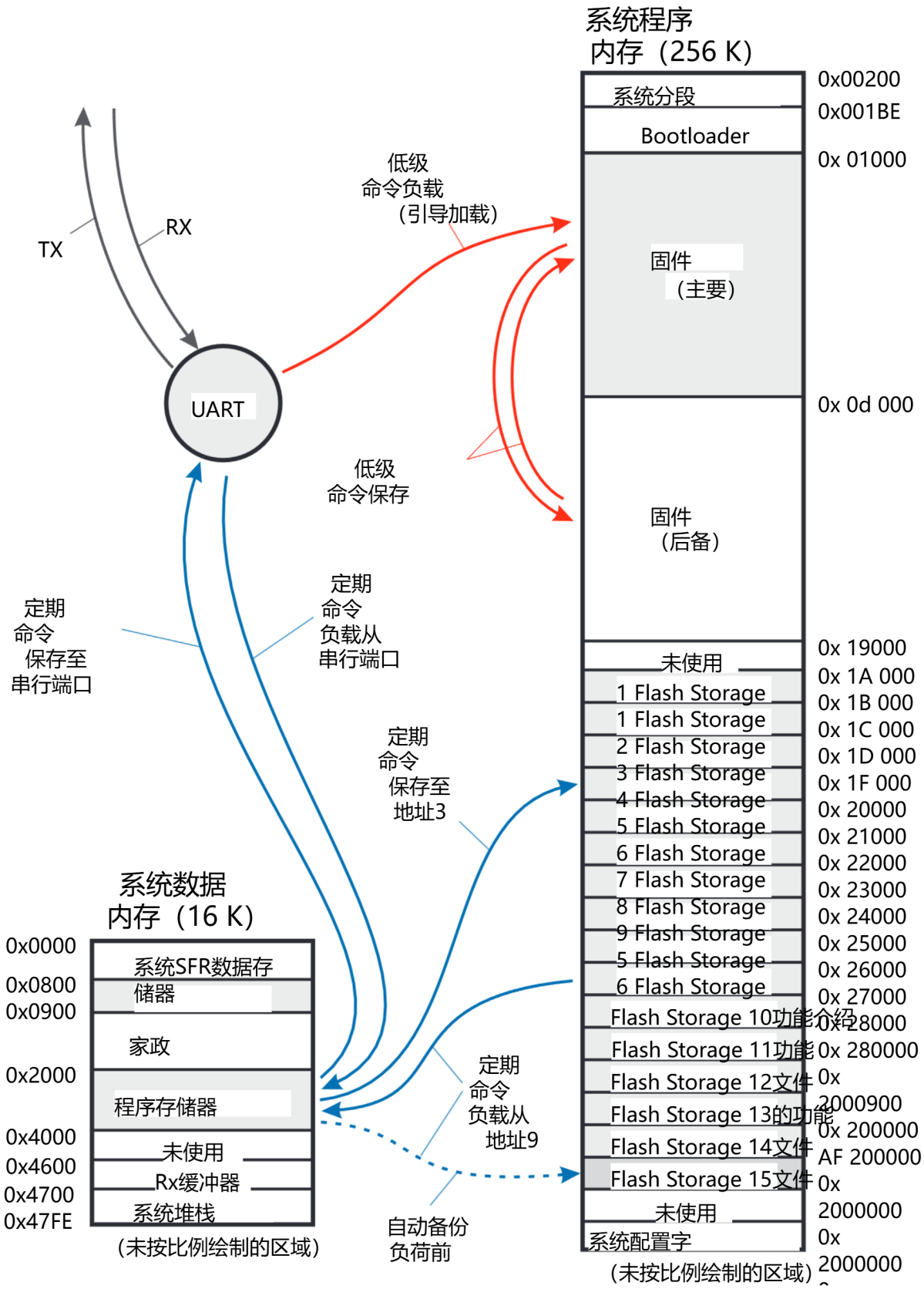
注：这是为假设模拟处理器的数据存储器表示，而不是系统处理器PIC 24FJ256GA 704。

Memory Organization (system processor)



Note: this is the representation of the Program Memory for the system processor PIC24FJ256GA704, not the hypothetical simulated processor.

内存组织（系统处理器）



注：这是系统处理器PIC 24FJ256GA 704的程序内存表示，而不是假设模拟处理器。