Automated Redstone Manual

by CD4017BE

mod version: 3.0.1

Contents

1 About this manual

Ingame there already is basic information about the blocks and items available in their tooltips if you press SHIFT. So this manual contains the more detailed information about how things work in Automated Redstone as well as giving some examples. But crafting recipes are not available here, use the Just Enough Items (JEI) mod to view them ingame.

Most screenshots used here are edited to make them fit better into the document, so don't worry if your ingame should look a bit different.

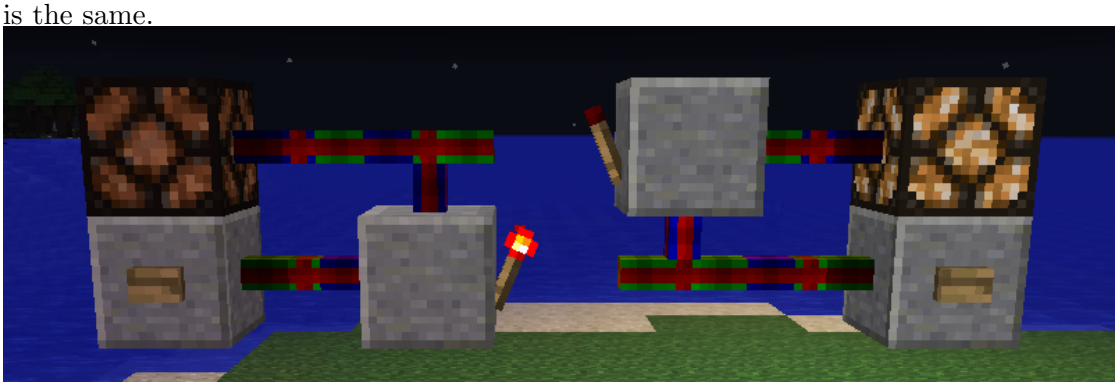
The information provided here refers to the mod version displayed above.

2 Signal transport

This mod adds a few cables to the game that allow transport of (redstone-) signals:

2.1 Analog Redstone cables

Analog redstone cables (currently named **1-bit solid redstone wire**) will receive redstone signals from their input connections (blue) and transmit them further to their output connections (green). If it has multiple inputs it will choose the strongest signal for output. To control the signal transfer direction, these cables come in three variants:
Input cables will by default connect to neighboring (non replaceable) blocks in the world and receive signals from these. They will try to establish output connections to other cables if possible.
Output cables also connect to other blocks and will emit their signal to them. They will try to establish input connections to other cables.
Transport cables only connect to other cables and will establish their connections so that they transport signals from input to output cables. They also connect to some other devices in this mod on their own. If you already used the basic item/fluid transport pipes from **Inductive Automation** you will notice that the connection behavior is the same.



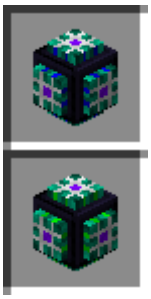
You can also sneak-right-click on cables to disconnect/reconnect them or cover them with solid opaque blocks by right-click (like in Inductive Automation).

The transmitted signal won't loose any strength and can be transported over theoretically unlimited distances. However there is a delay of one tick (50ms) every 16 blocks.

2.2 8-bit Redstone

8-bit redstone cables transmit 8 individual signal through one cable, where every signal can be either on or off (no strength in between). They only connect to other blocks that receive or emit 8-bit redstone and will orient their transfer direction similar to the analog transport cables. Disconnection and covering works the same like above.

2.3 8-bit wireless transfer

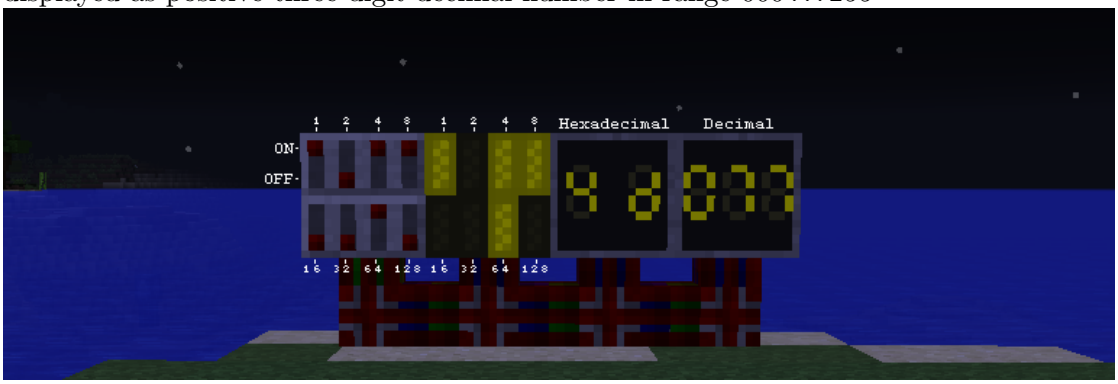


The 8-bit wireless connector comes in two variants: transmitter and receiver. By placing the block it will turn into a transmitter and a receiver block that is linked to it will be added to your inventory. If you break these blocks to place them somewhere else, they will stay linked to each other as long as you don't have both in item form at same time because they are just linked by knowing the position and dimension Id of the other block. If you manage to break the linkage, just craft them together into the original connector item again. Sneak-right-clicking a linked receiver / transmitter will remove both linked blocks simultanously and will also give you the original connector item back. (works like the interdimensional wormhole of InductiveAutomation)
The **8-bit wireless receiver** will emit the 8-bit singal that was received by its linked **8-bit wireless transmitter** to connected devices. The linkage also works across dimensions, but only if both components are chunkloaded.

3 Signal components

3.1 8-bit lever & display

An easy way to produce 8-bit signals is the **8-bit lever** . It has a front side with 8 red switches and will output its state to all connected 8-bit components.
And 8-bit states can be displayed using a **8-bit display** . It has three different display modes that can be changed by right-clicking the block's front:
In binary mode it will display the 8 states as individual lamps, arranged the same as the switches in the 8-bit lever are. In hexadecimal mode the state will be displayed as two digit hexadecimal number in range $00 \ldots FF$. And in decimal mode the state is displayed as positive three digit decimal number in range $000 \ldots 255$



3.2 Logic signal converter

The **logic signal converter** is used to combine or split multiple signals in digital from. In its gui you can set which sides of the block should be input or output and set the filter value that should be applied on each signal as AND-operation.

The internal logic of the block does the following:
At first all signals received at its input sides will be filtered. If it receives a 8-bit signal, it will be combined with the specified value using a logical AND-operation. Otherwise if it receives a normal redstone signal with a $stength > 0$ then the resulting 8-bit state will be equal to the specified filter value or 0 if $strength = 0$.
Then all filtered input states are combined with logical OR-operations.
Finally all output sides will emit the logical AND-operation between the resulting state and the individual filter value as a 8-bit signal and it will additionally emit a normal redstone signal of strength 15 if that 8-bit signal is not zero.

3.3 Arithmetic signal converter

The **arithmetic signal converter** is used to combine or split multiple signals in analog form. Here you can also set which sides should be input or output and set a multiplier that should be applied on each signal, as well as an offset value.

The internal logic of the block does the following:
At first all signals received at its input sided are converted into numbers and then multiplied with the specified multiplicator. For 8-bit signals it will use the represented binary number and for normal redstone signals it will use its $strength * 16$ (result: $0 \ldots 240$). The multiplication results and the specified offset value are added together. And finally all output sides will emit that sum divided by their specified multiplier with the result clipped to a range of $0 \ldots 255$. As 8-bit value that is just the binary representation of the number and as normal redstone value the strength will be the value divided by 16 rounded down.

4 Inventory control

4.1 Inventory connectors

4.2 Inventory reader

4.3 Item translocator

Documentation coming soon!

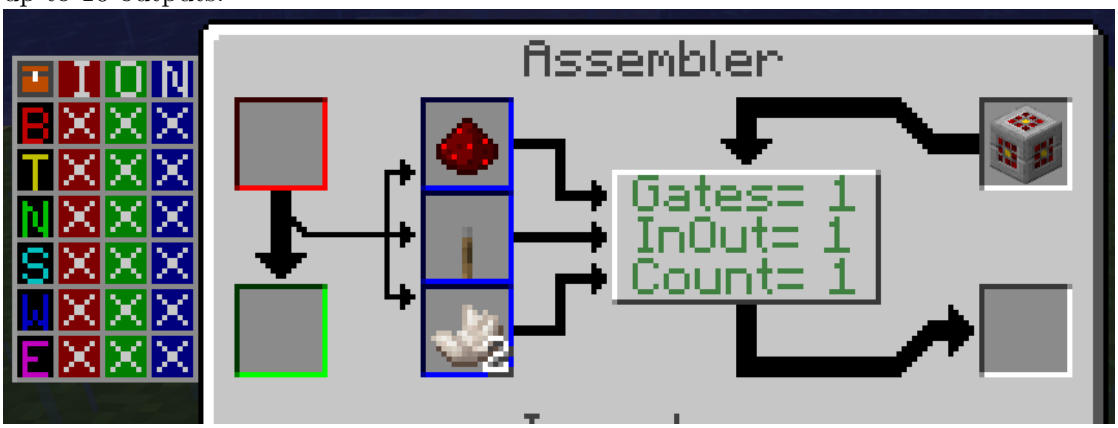
5 Integerated redstone circuits

The **Redstone Circuit** block allows you to put complicated redstone logic circuits into one single block, making them very compact and a bit cheaper in redstone cost. To define what the block should do you need to program it using the **Circuit Programmer** (explained later).

5.1 Creating & using a circuit

Also your circuit needs some logic gates, IO-ports, and counter to perform the program. These are installed using the **Circuit Assembler**. The exact amount of these parts depend on your program and is displayed in the Circuit Programmer when trying to put the program on the circuit.

For creating circuits in the assembler put the citcuit item into the top right slot. Adding logic gates requires 1 redstone dust each, adding counters requires 2 netherquartz each and adding available signal outputs requires 1 lever each. Put the ingredients into the 3 left slots and click on the big button int the middle to insert them. Circuits can also be disassembled again on the left side of the assembler's GUI giving you the parts back. The circuit's internal logic can contain up to 128 gates and/or up to 8 8-bit counter and up to 16 outputs.



About the colored tab on the very left read section 2.1 "Machine Configuration" in the Inductive Automation Manual.

When placed, the circuit block has 16 1-bit channels for each input and output, so to use all of them with only 6 available block faces, you would need to use 8-bit wire or place multiple circuits next to each other. Therefore they are bundled into four 8-bit chunks (2 in + 2 out).

In the circuits's GUI you can set for each block face, which 8-bit chunk to use and a HEX-AND-gate filter that should be applied (like in **logic signal converter**). So applying a normal redstone signal will set all bits contained in the filter to on and for output a redstone signal will be emitted if at least one of the bits contained in the filter is on.

The circuit's internal logic does the following: The state of all gates is updated each operation tick, based on the individually defined input states and the type of logic operation they represent. This is done in the order they appear in the programm using the input states present at that time, so referring to previous gates will use their just updated state and referring to later gates (or themself) will use the state they had after last update tick.

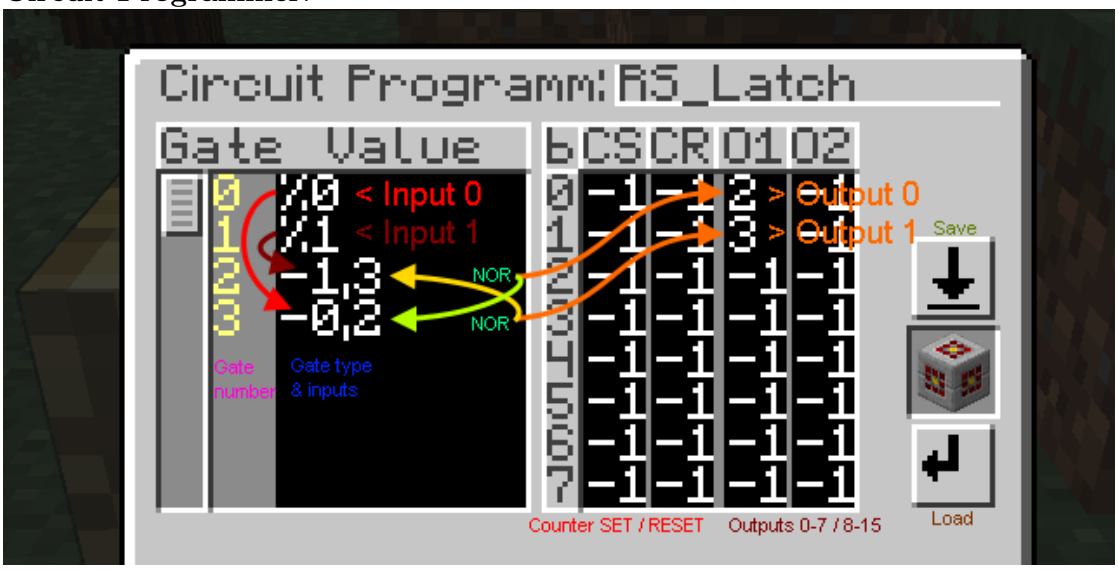
After that the counters will count up by one if the gate defined as SET-input is on, otherwise if the gate defined as RESET-input is on they will go back to 0. They also go back to 0 if they reach above 255. Their current 8-bit counting states are stored in gate slots 64 . . . 127 using 8 for each counter, so when using counters these slots are occupied and can't be used for logic gates.

Finally the circuit's outputs will be set to the state of the individual gates they are bound to.

The operation tick interval can be set in the circuit's GUI. It also contains a ON/OFF button to run or pause updating and a RESET button to set all gates to off.

5.2 Programming

To actually tell the circuit what it should do, you need to apply a program to it in the **Circuit Programmer**:



This shows it's GUI with a RS-Latch setup as example. The colored arrows and texts are edited to illustrate the signal flow.

Gates are defined on the left part of the GUI with the following syntax. Each line defines one gate and has to start with one character representing it's type and is directly (no whitespace) followed by its parameters.

For logic gates the parameters are the gate numbers that should be used as input, separated by colons ',' (also no whitespace). For redstone input it's the number of the circuit's input channel that this gate should be set to. And for the 8-bit comparator it's

the number of the 8-bit chunk of gates (= number of it's first contained gate divided by 8), whose numerical representation should be compared, followed by the comparator character, followed by a decimal number in range 0-255 to compare with. All currently available gate types are listed in this table:

Type	**Ch**	**Parameters**	**Function**
RS input	%	input channel	Active if channel receives redstone signal
OR gate	+	0-15 inputs	Active if any input active, otherwise inactive
NOR gate	−	0-15 inputs	Inactive if any input active, otherwise active
AND gate	&	0-15 inputs	Inactive if any input inactive, otherwise active
NAND gate	∗	0-15 inputs	Active if any input inactive, otherwise inactive
XOR gate	/	0-15 inputs	Active if uneven number of inputs active
XNOR gate	\	0-15 inputs	Active if even number of inputs active
8-bit Comp.	#	input=number	Active if 8-bit input equal to number
8-bit Comp.	#	input<number	Active if 8-bit input smaller than number
8-bit Comp.	#	input>number	Active if 8-bit input greater than number

The up to 8 counters of the circuit are defined in the columns on the right marked with 'CS' and 'CR'. The 'CS' (Counter SET) column gets the gate number that need to be active for the individual counter to count up by one. And the 'CR' (Counter RESET) column gets the gate number that needs to be active to set the counter back to zero. If reset gate is set to '−1' the counter will only go back to zero if it reaches an 8-bit integer overflow (> 255). Set both values to '−1' for counters you don't use (every counter costs 2 netherquartz).

As already mentioned before each counter occupies 8 gates to store its value, where counter 1 uses gates $64-71$, counter 2 uses $72-79$ and so on ($64+counterId*8$). These gates can be referenced in the program like any other gate, but the most common way is to use the 8-bit compare gate to check if the counter value is below, above or equal to a certain number. In this case the 8-bit chunk number for the comparator gate is simply the counter number plus 8.

And the last two columns marked with 'O1' and 'O2' define which gate should be used for which output channel, where 'O1' defines channels $0-7$ and 'O2' defines channels $8-15$. Set outputs to '-1' if you're not using them (every output costs a lever).

To put your settings onto a circuit, just give it a name in the line on top, put the circuit into the only available slot and click the save button. If everything is fine you should get a "Compiling successful" message and the tooltip of the circuit should show the program name you have set. If your gate definitions contain errors you get a message that tells "Compile error" with the line that was wrong. Otherwise if your circuit is missing gates, counter or outputs required to execute your settings you get a message that tells you what is missing.

To store your settings for later use you can also save it on a piece of paper and later load it into the programmer again by clicking the load button. If you don't need a circuit program anymore just put the item into the crafting grid to get the paper back.

5.3 Example programms

A **timer** that emits a 1 tick redstone signal every 150 ticks and resets if receiving a signal:

```
Gates:
0: %0              get input signal
1: #8=149          check if counter reached 149
2: +0,1            reset counter if it reached 149 or input active
3: -0              increase counter if input not active
Counter 0: CS=3 CR=2
Outputs: 1         emit signal if counter reached 149
Total cost: 4 redstone dust, 2 netherquartz, 1 lever
```

A **clock** that displays time in minutes and seconds on two 8-bit Displays. Input signal on channel 0 lets the clock run and signal on channel 1 resets it. (Circuit tick interval must be set to 0.05s):

```
Gates:
0: %0              get run signal
1: %1              get reset signal
2: #8=19           check if 19 ticks have passed
3: #9=59           check if 59 seconds have passed
4: &2,3            reset second counter after 59.95s
5: +1,4            or if receiving reset signal
6: +1,2            reset tick counter after 19 ticks or reset signal
Counter 0: CS=0 CR=6    counting ticks
Counter 1: CS=2 CR=5    counting seconds
Counter 2: CS=4 CR=1    counting minutes
Outputs:
72 - 77            to display seconds (6-bit)
80 - 87            to display minutes (8-bit)
Total cost: 7 redstone dust, 6 netherquartz, 14 lever
```

there will be probably added more in the future