



which C cross compilers are there for Atari ST and Amiga?



>

Cross compilers allow you to compile code for a target platform (such as the Atari ST or Amiga) on a different platform (such as a modern PC). Here are some C cross compilers that can be used for Atari ST and Amiga development:

For Atari ST:

1. GCC (GNU Compiler Collection) - This versatile compiler supports various platforms, including the Atari ST. You can use a version of GCC that targets the m68k architecture and Atari's TOS operating system.
2. AHCC (Pure C Compatible Compiler) - This is an Atari ST compatible C compiler that can be used for cross development on Windows and other platforms.

For Amiga:

1. vbcc (Volker Barthelmann's C Compiler) - This is a highly optimizing portable and retargetable ISO C compiler. It supports AmigaOS and several other platforms. You can find more information and downloads at the official vbcc website:
<http://www.ibaug.de/vbcc/>
2. GCC (GNU Compiler Collection) - As mentioned earlier, GCC can be used for Amiga development as well. You will need a version of GCC that targets the m68k architecture and AmigaOS.

For both Atari ST and Amiga, you'll need additional tools and libraries for development, such as SDKs, debuggers, and emulators. Additionally, you may want to explore various community resources and forums for more information on cross-development for these platforms.



what windows desktop C crosscompilers are for 8080



The Intel 8080 is an 8-bit microprocessor that was widely used in early personal computers and embedded systems. There are several C cross-compilers that you can use on Windows to develop software for the 8080 architecture:

1. Small Device C Compiler (SDCC): SDCC is a retargettable, optimizing Standard C (ANSI C89, ISO C99, ISO C11) compiler suite that targets various microprocessors, including the Intel 8080. It can be used on Windows, macOS, and Linux. You can find more information and downloads at the official SDCC website: <http://sdcc.sourceforge.net/>
2. z88dk: z88dk is a collection of software development tools, including a C compiler, that target the Zilog Z80 and Intel 8080 microprocessors. It's primarily designed for Z80-based systems, but it also supports the Intel 8080. z88dk is available for Windows, macOS, and Linux. You can find more information and downloads at the official z88dk website: <https://www.z88dk.org/>
3. CC80: CC80 is a simple and lightweight C compiler for the Intel 8080 and Z80 microprocessors. It runs on Windows and other platforms, and it can be used to generate 8080 machine code. You can find more information and downloads at the CC80 GitHub repository: <https://github.com/ncb85/cc80>

Remember that when working with cross-compilers, you may also need additional tools and libraries such as assemblers, linkers, debuggers, and emulators to aid in the development process.



is there some standard way how to redirect console in C to serial uart line?



In C, there is no standard way to redirect the console output to a serial UART (Universal Asynchronous Receiver/Transmitter) line, as the C standard library does not provide built-in support for serial communication. However, you can implement this functionality using platform-specific APIs.

Here's an example for Windows and POSIX-compliant systems (such as Linux and macOS):

1. Windows:

To redirect the console output to a serial port in Windows, you will need to use the Windows API functions for serial communication. Here's a basic outline of the process:

- a. Open the serial port using **CreateFile()**.
- b. Configure the serial port using **GetCommState()**, **SetCommState()**, and **SetCommTimeouts()**.
- c. Redirect the stdout to the serial port by using **_open_osfhandle()** and **_dup2()**.

Please note that this is a simplified description. For a detailed example, refer to the Microsoft documentation on serial communication: https://docs.microsoft.com/en-us/windows/win32/api/_fileio/

2. POSIX-compliant systems (Linux, macOS, etc.):

To redirect the console output to a serial port on POSIX systems, you can follow these steps:

- a. Open the serial port using **open()**.
- b. Configure the serial port using **termios** structures and **tcsetattr()**.
- c. Redirect the stdout to the serial port by using **dup2()**.

Here's a simple example in C for POSIX systems:

c

 Copy code

```
#include <fcntl.h> #include <stdio.h> #include <termios.h> #include <unistd.h>
void configure_serial_port(int fd) { struct termios options; tcgetattr(fd,
&options); cfsetispeed(&options, B115200); cfsetospeed(&options, B115200);
options.c_cflag |= (CLOCAL | CREAD); options.c_cflag &= ~PARENB; options.c_cflag
&= ~CSTOPB; options.c_cflag &= ~CSIZE; options.c_cflag |= CS8; tcsetattr(fd,
TCSANOW, &options); } int main() { int fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY
| O_SYNC); if (fd < 0) { perror("Unable to open serial port"); return 1; }
configure_serial_port(fd); // Redirect stdout to the serial port dup2(fd,
STDOUT_FILENO); printf("Hello, UART!\n"); close(fd); return 0; }
```

Replace `"/dev/ttyS0"` with the appropriate device file for your serial port. Note that the example uses a fixed baud rate of 115200 bps, but you can change this to match your requirements.

Please note that the example provided is a basic implementation. You may need to modify it depending on your specific use case and target platform.



okay, I am more interested by embedded solutions - so I must implement this custom way? its possible to use in code regular stdio ??



Yes, you can still use the regular **stdio** functions in embedded systems, but you will need to provide custom implementations of low-level I/O functions that interface with your specific hardware. This usually involves implementing functions like `_write()` and `_read()`.

For example, if you are using an ARM Cortex-M microcontroller with the STM32 series, you can redirect the standard output (`printf()`) to a UART by implementing the `_write()` function provided by the C standard library.

Here's a simple example for STM32 using the STM32Cube HAL library:

c

Copy code

```
#include "stm32f1xx_hal.h" #include "stm32f1xx_hal_uart.h" #include <errno.h>
#include <sys/unistd.h> // for STDOUT_FILENO extern UART_HandleTypeDef huart1;
// Assuming you have already initialized huart1 int _write(int file, char *data,
int len) { if (file != STDOUT_FILENO) { errno = EBADF; return -1; }
HAL_StatusTypeDef status = HAL_UART_Transmit(&huart1, (uint8_t *)data, len,
100); return (status == HAL_OK) ? len : 0; }
```

This example assumes that you have already initialized the UART with the proper configuration using the STM32Cube HAL library (e.g., `MX_USART1_UART_Init()`). With this implementation, you can use `printf()` to send data over UART.

Please note that this example is specific to STM32 microcontrollers using the STM32Cube HAL library. The exact implementation will depend on the microcontroller you are using and the libraries or hardware abstraction layers you have chosen.

To implement this custom way, you should refer to the documentation and libraries provided by the manufacturer of the microcontroller you are using. They usually provide

examples and guidelines on how to interface with the UART peripheral and other hardware components.



is something like `_write` and `_read` there also for PIC16F 18F 24F 32 ??



Yes, similar to the ARM Cortex-M example, you can redirect the standard I/O functions like `printf()` and `scanf()` to a UART peripheral for PIC microcontrollers (PIC16F, PIC18F, PIC24F, and PIC32) using the MPLAB XC8, XC16, and XC32 compilers provided by Microchip.

You will need to implement the `_mon_putc()` and `_mon_getc()` functions for XC8 (PIC16F and PIC18F) and the `_write()` and `_read()` functions for XC16 (PIC24F) and XC32 (PIC32).

Here's an example for each series of microcontrollers:

1. PIC16F/PIC18F (XC8 Compiler):

c

 Copy code

```
#include <xc.h> #include <stdio.h> #include "uart.h" // Assuming you have a UART
library with UART_Write and UART_Read functions void _mon_putc(char c) {
UART_Write(c); } char _mon_getc() { return UART_Read(); }
```

2. PIC24F (XC16 Compiler):

c

 Copy code

```
#include <xc.h> #include <stdio.h> #include <errno.h> #include <sys/unistd.h>
#include "uart.h" // Assuming you have a UART library with UART_Write and
UART_Read functions int _write(int file, char *buffer, int len) { if (file !=
STDOUT_FILENO) { errno = EBADF; return -1; } int count = len; while (count--) {
UART_Write(*buffer++); } return len; } int _read(int file, char *buffer, int
len) { if (file != STDIN_FILENO) { errno = EBADF; return -1; } int count = len;
while (count--) { *buffer++ = UART_Read(); } return len; }
```

3. PIC32 (XC32 Compiler):

```
#include <xc.h> #include <stdio.h> #include <errno.h> #include <sys/unistd.h>
#include "uart.h" // Assuming you have a UART library with UART_Write and
UART_Read functions int _write(int file, char *buffer, int len) { if (file !=
STDOUT_FILENO) { errno = EBADF; return -1; } int count = len; while (count--) {
UART_Write(*buffer++); } return len; } int _read(int file, char *buffer, int
len) { if (file != STDIN_FILENO) { errno = EBADF; return -1; } int count = len;
while (count--) { *buffer++ = UART_Read(); } return len; }
```

For each example, you will need to provide the `UART_Write()` and `UART_Read()` functions that interact with the UART peripheral on your specific PIC microcontroller. You can refer to the Microchip documentation, libraries, and examples provided in the MPLAB X IDE to implement these functions.

After implementing the corresponding functions for your PIC microcontroller and compiler, you can use the standard I/O functions like `printf()` and `scanf()` to send and receive data over UART.



how about arduino here?



On Arduino platforms, you don't need to implement low-level I/O functions like `_write()` or `_read()`, as the Arduino libraries already handle the abstraction for you. Instead, you can use the `Serial` object to interact with the UART.

Here's an example of using UART with an Arduino board:

1. First, include the Arduino library and initialize the `Serial` object in the `setup()` function:

cpp

 Copy code

```
#include <Arduino.h> void setup() { // Initialize serial communication at a baud
rate of 9600 bps Serial.begin(9600); // Wait for the serial port to be ready
while (!Serial) { ; // Wait for the serial port to connect } }
```

2. Then, use `Serial.print()`, `Serial.println()`, and `Serial.write()` functions to send data over UART, and `Serial.read()`, `Serial.readBytes()`, `Serial.available()`, and other related functions to receive data:

cpp

 Copy code

```
void loop() { // Send data over UART Serial.println("Hello, UART!"); // Check if
there's data available to read if (Serial.available() > 0) { // Read a character
from the UART char receivedChar = (char)Serial.read(); // Echo the received
character back Serial.print("Received: "); Serial.println(receivedChar); } //
Add a delay between iterations delay(1000); }
```

The `Serial` object in the Arduino library abstracts away the low-level details, making it simple to work with UART communication. With this approach, you don't need to worry about implementing functions like `_write()` or `_read()` as you would with other microcontrollers.



yeah, but in arduino is no stdio supported, ya?



You're correct that Arduino does not directly support the standard C I/O library (`stdio`) by default. However, you can still use `printf()` and related functions by implementing a custom `write()` function and wrapping the `Serial` object into a `FILE` stream.

Here's an example of how to use `printf()` with the Arduino `Serial` object:

1. Include the necessary libraries:

cpp

 Copy code

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

2. Implement a custom `write()` function that wraps the Arduino `Serial.write()` method:

cpp

 Copy code

```
static int uart_putchar(char c, FILE *stream) { Serial.write(c); return 0; }
```

3. Create a **FILE** stream that uses the custom **write()** function:

cpp

 Copy code

```
static FILE uartout = {0};
```

4. In the **setup()** function, configure the **Serial** object and redirect the **stdout** to the custom **FILE** stream:

cpp

 Copy code

```
void setup() { // Initialize serial communication at a baud rate of 9600 bps  
Serial.begin(9600); // Wait for the serial port to be ready while (!Serial) { ;  
// Wait for the serial port to connect } // Redirect stdout to the custom FILE  
stream fdev_setup_stream(&uartout, uart_putchar, NULL, _FDEV_SETUP_WRITE);  
stdout = &uartout; }
```

5. Now you can use **printf()** in the **loop()** function or elsewhere in your code:

cpp

 Copy code

```
void loop() { printf("Hello, UART! Value: %d\n", 42); // Add a delay between  
iterations delay(1000); }
```

This example demonstrates how to implement a custom **write()** function and use it with the Arduino **Serial** object to support **printf()**. Note that this approach does not provide support for **scanf()** and other input-related functions from **stdio**. If you need input functionality, you can create a similar custom **read()** function and wrap the Arduino **Serial.read()** method to achieve that.





what you think about this license?:

1.2 Legal

vbcc is copyright in 1995-2022 by Volker Barthelmann.

This archive may be redistributed without modifications and used for non-commercial purposes.

An exception for commercial usage is granted, provided that the target CPU is M68k and the target OS is AmigaOS. Resulting binaries may be distributed commercially without further licensing.

2 vbcc manual

An exception for commercial usage is granted, provided that the target CPU is 6502 with MEGA65 extensions and the target HW is MEGA65. Resulting binaries may be distributed commercially without further licensing.

In all other cases you need my written consent.

This copyright applies to vc, vbcc, vsc and vcpr.

This archive may contain other tools (e.g. assemblers or linkers) which do not fall under this license. Please consult the corresponding documentation of these tools.

vbcc contains the preprocessor ucpp by Thomas Pornin. Included is the copyright notice of ucpp (note that this license does not apply to vbcc or any other part of this distribution):

/*

* (c) Thomas Pornin 1999, 2000

*

* Redistribution and use in source and binary forms, with or without

* modification, are permitted provided that the following conditions

* are met:

* 1. Redistributions of source code must retain the above copyright

* notice, this list of conditions and the following disclaimer.

* 2. Redistributions in binary form must reproduce the above copyright

* notice, this list of conditions and the following disclaimer in the

* documentation and/or other materials provided with the distribution.

* 4. The name of the authors may not be used to endorse or promote

* products derived from this software without specific prior written

* permission.

*

* THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR

* IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED

* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE

* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR

* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT

* OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
* BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
* WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
* OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
* EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*
*/



The given license appears to be a custom license with certain conditions and exceptions for usage. Here's a summary of its main points:

1. The copyright for vbcc is held by Volker Barthelmann from 1995 to 2022.
2. Redistribution of the archive is allowed without modifications and for non-commercial purposes.
3. Exceptions for commercial usage are granted for:
 - a. Target CPU: M68k and Target OS: AmigaOS
 - b. Target CPU: 6502 with MEGA65 extensions and Target HW: MEGA65In these cases, the resulting binaries can be distributed commercially without further licensing.
4. For other cases, written consent from the copyright holder is required.
5. The copyright applies to vc, vbcc, vsc, and vcpr tools.
6. The archive may contain other tools not covered by this license. Consult the corresponding documentation for these tools.
7. vbcc contains the preprocessor ucpr by Thomas Pornin, which has its own copyright notice and license. This license does not apply to vbcc or any other part of the distribution.

As an AI language model, I am not able to provide legal advice, but this summary should help you understand the main points of the license. If you have concerns about using the software under this license, you should consult with a legal professional for guidance.





tnx



You're welcome! If you have any more questions or need further assistance, please don't hesitate to ask. Happy coding!



how can this compiler VBCC be used to compile vmex.c source for atari-st and amiga?



To compile a C source file (e.g., **vmex.c**) for Atari ST and Amiga using the VBCC compiler, you'll need to follow these steps:

1. Download and install the VBCC compiler: You can download VBCC from the official website (<http://www.ibaug.de/vbcc/>). Extract the archive and follow the installation instructions provided in the documentation.
2. Download and install the appropriate target-specific files for Atari ST and Amiga. VBCC provides separate target-specific files for different systems. Download the files for Atari ST (usually named **vbcc_target_m68k-atari.zip**) and Amiga (usually named **vbcc_target_m68k-amiga.zip**), and extract them to the appropriate directories as per the VBCC documentation.
3. Set up the environment variables: You need to set the **VBCC** environment variable to point to the root directory of your VBCC installation, and the **PATH** environment variable to include the **bin** directory of your VBCC installation. This can be done in the terminal or command prompt, depending on your operating system.
4. Compile the C source file for Atari ST:

```
sh
```

Copy code

```
vc -I<path_to_atari_st_target_headers> -c99 -O2 -+ -target=m68k-atari-tos vmex.c  
-o vmex_atari_st
```

Replace `<path_to_atari_st_target_headers>` with the path to the include directory containing the Atari ST target-specific header files.

5. Compile the C source file for Amiga:

sh

 Copy code

```
vc -I<path_to_amiga_target_headers> -c99 -O2 -+ -target=m68k-amigaos vmex.c -o  
vmex_amiga
```

Replace `<path_to_amiga_target_headers>` with the path to the include directory containing the Amiga target-specific header files.

These commands will compile the `vmex.c` source file for Atari ST and Amiga, producing the output binaries `vmex_atari_st` and `vmex_amiga`, respectively. Note that you might need to adjust the compiler flags and options depending on your specific requirements and the nature of the source code.



it reports "no config file"



Apologies for the oversight. The "no config file" error occurs when the VBCC compiler is unable to find the configuration file for the target system. To resolve this issue, you need to set up the `vc` command with the correct configuration file for your target platform (Atari ST or Amiga).

1. Locate the configuration files `vc.config.tos` (for Atari ST) and `vc.config` (for Amiga) in the respective target-specific directories you extracted in the previous steps. They are usually located in the root directory of each target-specific folder.
2. Set the `VC_CONFIG_DIR` environment variable to point to the directory containing the desired configuration file. For example, if you want to compile for Atari ST, set the `VC_CONFIG_DIR` variable to the directory containing `vc.config.tos`.
3. Compile the C source file for Atari ST:

sh

 Copy code

```
vc -c99 -O2 -+ -config=vc.config.tos -I<path_to_atari_st_target_headers> vmex.c  
-o vmex_atari_st
```

4. Compile the C source file for Amiga:

```
sh
```

 Copy code

```
vc -c99 -O2 -+ -config=vc.config -I<path_to_amiga_target_headers> vmex.c -o  
vmex_amiga
```

Make sure to replace **<path_to_atari_st_target_headers>** and **<path_to_amiga_target_headers>** with the paths to the respective include directories containing the target-specific header files.

With the **-config** option and the **VC_CONFIG_DIR** environment variable properly set, the "no config file" error should be resolved, and the compiler should be able to produce the output binaries for Atari ST and Amiga.
