Toolchain and workflow for the design of an ISO 11783-compatible ECU based on ISOAgLib

Joaquin Ezpeleta and Sebastián Rossi,

Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario,
Av. Pellegrini 250, S2000BTP Rosario, Argentina
ezpeleta@fceia.unr.edu.ar, srossi@inti.gob.ar

Abstract. This paper describes a basic toolchain for the design of an ISO 11783-compatible electronic control unit (ECU), from its conception to the implementation of a working embedded prototype, along with a suggested workflow for dividing application programming, mask design and hardware-related tasks in a debugging-friendly and time-efficient manner. The toolchain is centered on the open source ISOAgLib programming library distributed and maintained by OSB AG and the paper will refer to other specific tools and devices, but is otherwise intended to provide a general introductory overview of the process rather than focus on specific vendors or platforms.

Keywords: toolchain, workflow, ISO 11783, ISOBus, ISOAgLib, controller area network (CAN), embedded system.

1 Introduction

ISO 11783 [1 *et seq.*] –commonly known as ISOBus– defines a serial data network for agricultural or forestry equipment based on the CAN 2.0 B [4] protocol. It is intended to provide an interconnection system for on-board electronics. A typical ISOBus network is shown on Figure 1. A basic element of such network is the electronic control unit (ECU), which is defined in [1] as an 'electronic item consisting of a combination of basic parts, subassemblies and assemblies packaged together as a physically independent entity'.

This paper describes the process for creating one such ECU, from its design to the implementation of an embedded prototype. Specifically, it suggests a toolchain and a workflow for this process. The toolchain is based on the open-source library ISOAgLib and other related tools distributed and maintained by OSB AG [7], but alternatives are given wherever possible to provide maximum flexibility. Similarly, the process described is mostly platform-independent (both in terms of the operating system used for development on a desktop environment and in terms of the embedded hardware platform used for the actual implementation), but examples are given at various points for the purpose of clarity and reproducibility. For an implementation on specific hardware see, for example, [6].

Section 2 presents the suggested toolchain, along with explanations and brief examples; Section 3 describes the physical and virtual elements needed to test and use

software- and firmware-level applications; Section 4 presents the proposed workflow for an efficient and debugging-friendly task division using the tools and resources presented in Sections 2 and 3.

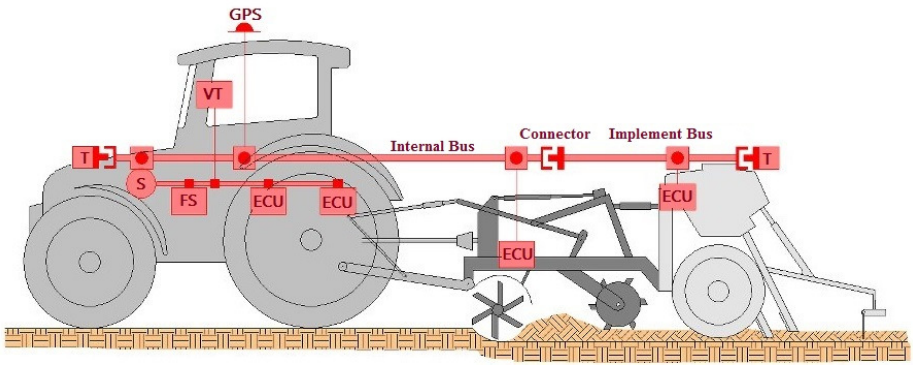

Fig. 1. Example ISO 11783 network on a tractor with two implements.

2 Toolchain

A summary of the suggested toolchain is shown graphically in Figure 2. It presents the tasks needed to design an ISOBus system with ISOAgLib, the tools needed for each task and the relationship between these tasks. As will be seen on Section 4, however, many of these tasks can often be undertaken independently, so a roughly left-to-right and top-to-bottom order will be followed.

2.1 Mask design and parsing

The first tool to be discussed is VT designer, which is a graphic interface for designing masks which are compatible with [3]. It is currently available from OSB AG in both a trial and a full version. It has a simple and intuitive interface and includes examples which can be modified to gain insight into its use. The purpose of this tool within the suggested toolchain is to take an abstract design concept for the necessary mask(s) and create files which can be further processed for use in the application. Given one or more manually-loaded masks (collectively, an object pool), VT Designer creates a .vtp project file and a number of .xml files which contain the actual objects and attributes.

An alternative for the use of VT Designer is PoolEdit, developed by Matti Öhman and Jouko Kalmari at Aalto University [9] and distributed under GNU General Public License (GPL). The former is discussed here for its greater affinity with the OSB AG toolchain, but the latter produces almost identical results.

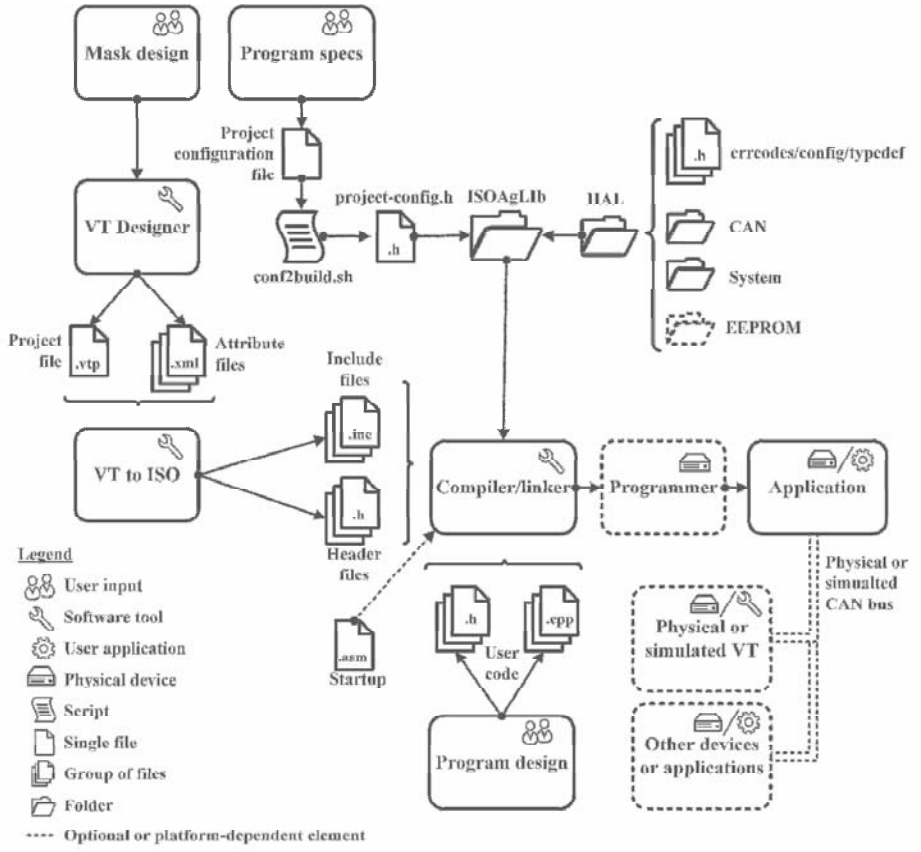

Fig. 2. Toolchain for the development of an ISO 11783-compatible application.

The .xml files output by VT Designer cannot typically be included directly into the source code of the application. A second tool is needed to transform the .xml files into source files which can be handled by the C++ compiler. An example of an application which serves this purpose is vt2iso.exe, available free of charge from OSB AG. It is a console application which essentially takes the .vtp and .xml files from the previous step and produces a group of .inc and .h files which can then be included directly by the compiler. Below is a basic use example on the Microsoft Windows command line, where myDisplay.vtp is a VT Designer project file. A number of optional arguments are also available but are not typically necessary.

```
C:\IsoAgLib\tools\vt2iso\bin>vt2iso.exe myDisplay.vtp
```

2.2 Project Configuration

Another step in the design process is defining project settings. These are to be entered manually in a configuration file and include basic parameters, such as the project name, the folder where the user and library files are stored, the number of CAN instances and the target platform (e.g. PC running Microsoft Windows, PC running Linux, embedded system, etc.), among others.

It is advisable to adapt the template (conf_template) included with ISOAgLib rather than enter the settings directly on a blank file, as the former approach will be both faster and less error-prone. In either case, the resulting file is to be processed by a shell script (conf2build.sh) to produce a configuration header (isoaglib_project_config.h) which can be included directly into the application.

The shell script is included with ISOAgLib and can be run natively in Linux and other UNIX systems. Microsoft Windows users will need to install MSYS/MINGW [11] or other similar software which makes it possible to run UNIX shell scripts under this OS. A typical command for executing the script is shown below, where myconfig is a copy of conf_template adapted to a specific project.

```
$ conf2build.sh myconfig
```

Also note that the resulting isoaglib_project_config.h header is included via the library header isoaglib_config.h and is not to be included directly (the preprocessor will issue an error if it is).

2.3 Hardware Abstraction Layer (HAL)

ISOAgLib includes a Hardware Abstraction Layer (HAL) which acts as an intermediary between the rest of ISOAgLib and the actual hardware on which it runs. This includes system startup, time tracking, power management, CAN communication, non-volatile storage and similar hardware-dependent functions, along with definitions for data types, error codes and configuration parameters.

The HALs for certain hardware platforms (most notably, PCs running Microsoft Windows or Linux) are already included with the library, but others must be programmed by the user (e.g. ARM-based MCUs). It is advisable to use an existing HAL as model when creating a new one. It should also be borne in mind that, when changing platforms, the project configuration described in 2.2 must be updated accordingly.

In embedded systems, it may also be necessary to include additional files outside the HAL, such as a startup file (see Figure 2, bottom left).

2.4 User Code

While ISOAgLib provides for most communication and compliance requirements, it is up to the user to design and program the actual application. The typical application includes initialization code which is run once at the start, a main application loop

which performs all communication and normal operation tasks and closing code which shuts down the application and all associated hardware in a controlled manner. In addition, on embedded systems, startup code is typically needed before the rest of the application can be executed.

The initialization code should perform the following basic functions: (a) initialize instances of the system, the scheduler, the bus, the monitor and the CAN controller; (b) declare or retrieve parameters needed for subsequent address claim; (c) register the necessary tasks on the scheduler, initialize the system components and register the object pool on the monitor; (d) transfer control to the main loop. In addition, it should perform any application-specific or hardware-dependent initialization functions (such as interrupt vector, watchdog, real-time clock, analog-digital converter or output configuration or pin remapping), although these can usually be called automatically during system initialization by including them in the HAL.

The main application loop should run after initialization and until the application needs to terminate for any reason. As far as the ISOAgLib library is concerned, the only requirement for the main application loop is that the time event of the scheduler instance be called within the time constraints defined in [1 *et seq.*], but other functions will be needed for any specific application.

Finally, the closing code unregisters all elements, deallocates memory, calls the close method for each of the active instances and takes all associated hardware to a safe condition (e.g. deenergized actuators).

Gaining insight into the use of ISOAgLib can be challenging given the lack of freely available training and tutorials. OSB AG offers customized workshops and training sessions which include examples, but the examples themselves cannot be bought alone. If workshops or training are not an option due to cost, examples are freely available for earlier versions of ISOAgLib (up to 2.2.1) from OSB AG's repository. These can be adapted to more current versions by following the changelog and inferring necessary changes in the user files, but this involves considerable guesswork and can be a time-consuming process.

2.4 Compilation, Linkage and Loading

The tools used for compilation, linkage and (in the case of embedded systems) loading are highly platform-dependent and are usually managed by a single IDE. General purpose compilers such as GCC [10] can be used for early stages of development where the application is to run entirely within the desktop environment, while vendor- or platform-specific compilers are generally needed to compile the application for specific hardware (e.g. armcc for ARM-based MCUs).

Remark 1. ISOAgLib is written in C++. This must be taken into account when selecting the product and compiler, as some products do not offer C++ compilers or do so at relatively high prices.

3 Hardware framework

Figure 3 shows an example interconnection of system elements. It is intended to provide a summary of the connection options available throughout the entire design process. In each stage of the development proces, however, not all of the elements are simultaneously necessary in general. For example, while initially programming and debugging the application, all work is typically done within a desktop environment without resorting to additional external hardware. Similarly, the process for preliminary design and implementation of sensors and actuators will not normally require access to a virtual terminal, whether physical or simulated.

The interconnection between system elements is done basically by means of three CAN buses, namely a physical bus, a socket bus and a proprietary virtual bus.

The physical bus is an actual wired bus complying with the requirements set forth in [2]. It includes a pair of data wires CAN_H and CAN_L and should be terminated with impedance-matching resistors.

Remark 2. While [2] defines a nominal characteristic bus impedance of 75 Ω, existing CAN hardware not developed specifically for ISOBus may use 120 Ω instead, as defined in [5] (the original Bosch document, [4], did not specify this and other electrical aspects of the physical layer).

Data transmission from and to the bus is normally done through CAN transceivers. These convert 3.3 V or 5 V logic values from the RX/TX pins on an MCU (or other levels from other hardware) to dominant and recessive bits on the CAN bus and vice versa. Examples of CAN transceivers are SN65HVD230 and MAX3051 for 3.3 V systems and SN65HVD255 and MAX3058 for 5 V systems.

The socket bus is an internal socket connection which emulates a CAN bus within a desktop environment. The ISOAgLib toolchain includes several similar tools for this purpose, one which is designed solely for interaction between several ISOAgLib applications within the desktop (can_server_simulating.exe) and others which additionally provide functionality for connection with vendor-specific hardware and software. For example, can_server_vector.exe and can_server_vector_xl.exe are designed for communicating with hardware and software from Vector Informatik GmbH [8], which is achieved by using Vector's CAN DLL library. The CAN server thereby acts as a bridge between the socket bus and the proprietary virtual bus, discussed below.

The proprietary virtual bus is an emulated CAN bus like the socket bus, and can be thought of as an extension of the latter. It is used for communication between proprietary hardware and software from a given vendor and the corresponding CAN server. In addition, by communicating with a CAN card, the proprietary virtual bus in turn enables access to the physical external bus. For example, Vector offers CANcardXL, a two-channel driver card which provides access to up to two external physical buses. Additionally, they offer an application by the name of CANoe which manages the card and also serves as a bus analyzer. When both CANcardXL and can_server_vector_xl.exe are running, the socket bus, the proprietary virtual bus and the physical bus merge, for all practical purposes, into a single CAN bus. This and other similar setups offer great versatility, as they make it possible for a number of

applications running either on the desktop or on an embedded system to communicate seamlessly with each other and with third-party devices, such as a virtual terminal. In addition, software like CANoe can emulate a virtual terminal, which is essential for running preliminary tests entirely within the desktop environment, with further tests on actual virtual terminals being done at a later stage of development.

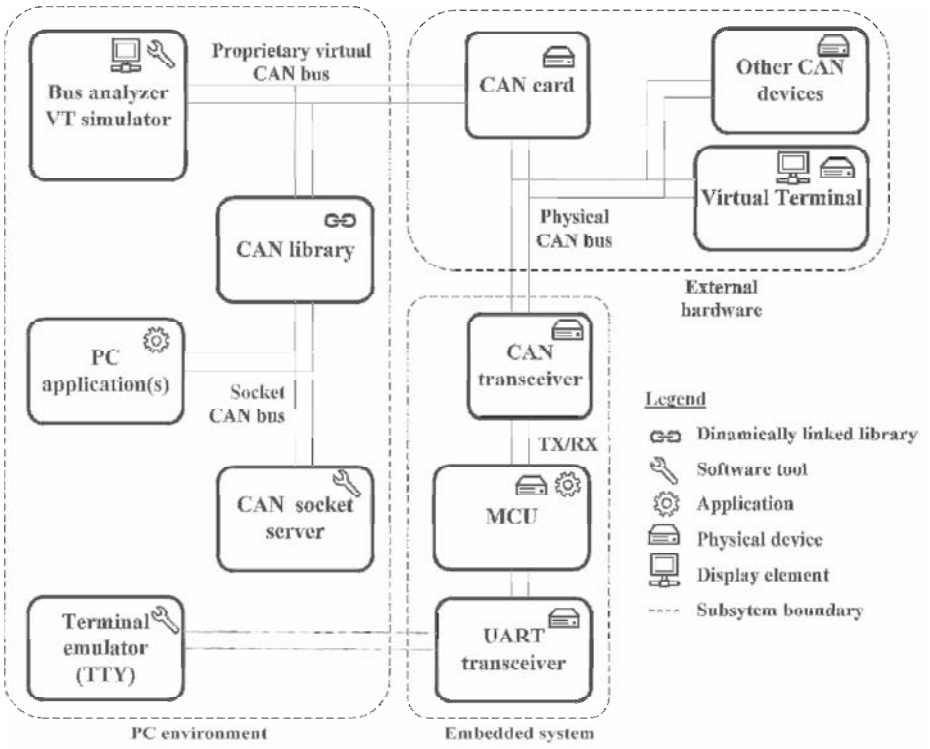

Fig. 3. Hardware framework.

In addition to the CAN buses, Figure 3 shows a UART connection between the MCU and a terminal emulator within the PC. This provides a simple means of transmitting debugging information between the PC and the MCU without loading the CAN bus, but is entirely optional.

4 Workflow

This Section describes a proposed workflow for the design process of an ISO 11783-compatible application. The basic stages in this process are application programming (with and without masks), mask design, hardware-related tasks and prototyping. A way of organizing these stages is shown in Figure 4. The basic concept is to run tasks in parallel in order to make better use of available time, hardware and

human resources. Time is better used as different teams can work on the different task simultaneously. In addition, each task can be assigned to a specialist in the corresponding field, such as a programmer for building the application and an electrical engineer for working wiring, sensors and actuators, making better use of available human resources. Finally, available hardware can be used more efficiently, as each task requires only some of the hardware tools. Some examples of this were discussed in the previous Section, which described a flexible hardware framework.

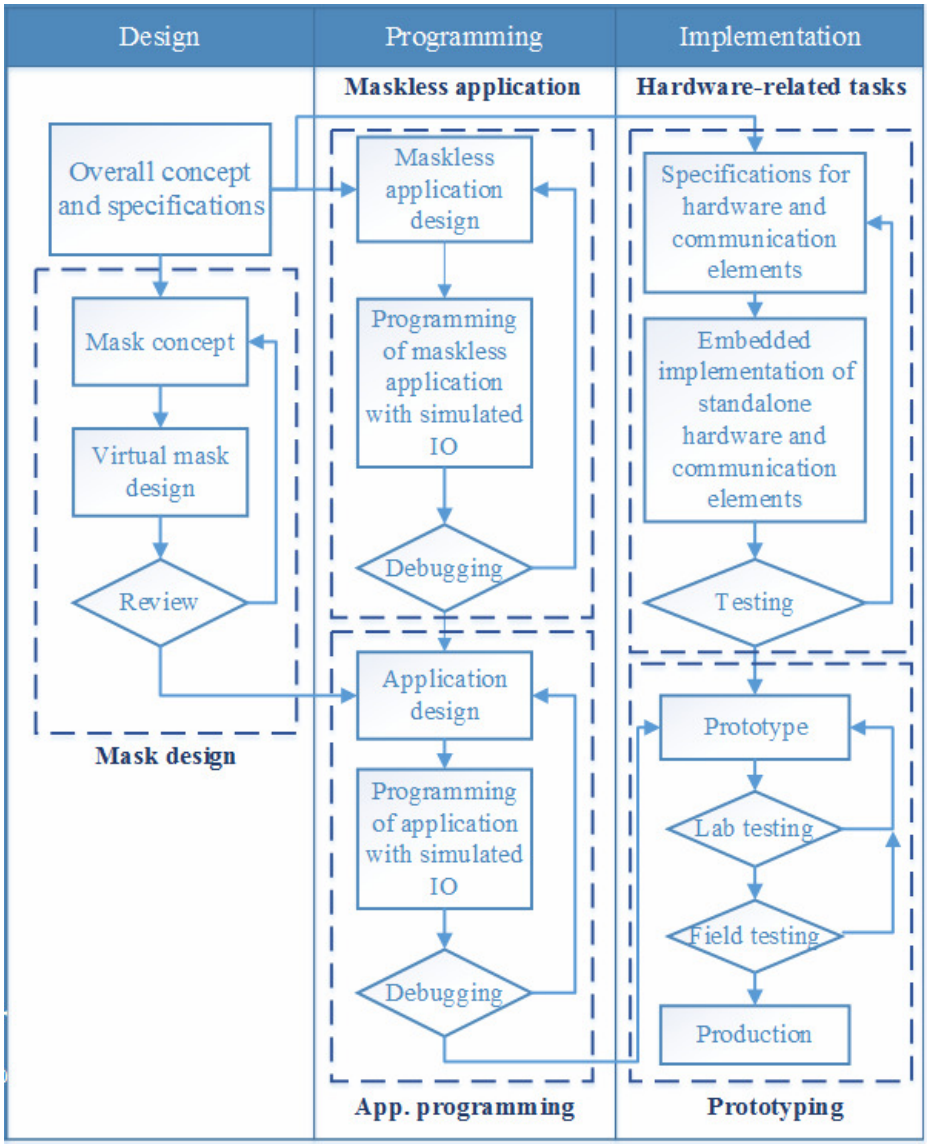

Fig. 4. Proposed workflow for the design process.

Another advantage of task division is to facilitate the debugging and troubleshooting process. When each task is undertaken individually, the possible error sources are confined to a subdomain of the entire system. For example, if the application is initially debugged using a virtual bus and simulated data (maskless application or application programming stage in Figure 4), no communication- or hardware-related errors arise and the debugging domain is thereby restricted only to the application itself.

The maskless application programming stage consists in the design and programming of a virtual application which does not resort to any virtual terminal (physical or otherwise) for display. Additionally, such application would be developed entirely within a desktop environment, without resorting to external data or elements and without being loaded onto an MCU or other embedded system. The application uses the PC HAL at this point. As the application will be isolated from data which it normally needs for its operation (data from sensors, other network devices or user input), simulated data can be used during this stage to examine the behavior of the application in different situations.

The mask design stage basically involves the task of designing one or more masks (an object pool) and parsing then into a format which can be handled directly by the compiler used for building the application. This stage can be completed mostly using the tools described in 2.1 above, although some previous design work is necessary to create a visual concept for the masks (colors, layout, shape and position of various elements, expected functionality, etc.).

The application programming stage is an extension of the maskless application programming stage with the addition of the masks designed in the mask design stage. The goal of this stage is to ensure that the application can upload the object pool to a simulated virtual terminal and interact with it (i.e. receive user input and make necessary changes on the elements of the object pool). Except for data flowing to and from the virtual terminal, the rest of the data used during this stage is still simulated. Similarly, the application continues to run within the desktop environment using a PC HAL.

The hardware-related tasks stage encompasses all hardware or platform-specific tasks. These basically include the design and implementation of actuators, drivers, sensors, data acquisition means, the creation of a HAL for ISOAgLib if one does not exist for the intended platform and the inclusion or programming of other platform-dependent elements which cannot be included within the HAL (such as a startup file). The creation of the HAL in turn involves implementing functions for CAN communication, configuration of interrupt vectors, watchdogs, real-time clock, analog-digital converters or outputs and pin remapping.

Finally, the application (included the masks) can be loaded onto the embedded hardware and combined with sensors, actuators, CAN peripherals and other hardware elements to produce a working embedded prototype. This prototype can then be tested for minor bugs or problems within a laboratory environment (possibly using a simulated VT) to create a final version of the device, which can be further tested with actual equipment on field prior to its commercial production.

References

1. ISO 11783-1:2007, Tractors and machinery for agriculture and forestry — Serial control and communications data network — Part 1: General Standard for mobile data communication, International Organization for Standardization, Geneva (2007)
2. ISO 11783-2:2002, Tractors and machinery for agriculture and forestry — Serial control and communications data network — Part 2: Physical Layer, International Organization for Standardization, Geneva (2002)
3. ISO 11783-6:2004, Tractors and machinery for agriculture and forestry — Serial control and communications data network — Part 6: Virtual Terminal, International Organization for Standardization, Geneva (2004)
4. CAN Specification Version 2.0 Part B, Robert Bosch GmbH, Stuttgart (1991)
5. ISO 11898-2:2003, Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit, International Organization for Standardization, Geneva (2003)
6. Tumenjargal, E., Badarch, L., Kwon, H., Ham, W.: Embedded software implementation system for a human machine interface based on ISOAgLib. Journal of Zheijang University (2013)
7. OSB AG, http://www.osb-ag.com/osb-ag.html
8. Vector Informatik GmbH, http://vector.com
9. PoolEdit - Open Source XML ISO 11783 User Interface Editor, http://autsys.aalto.fi/en/Farmix/PoolEdit
10. GCC, the GNU Compiler Collection, http://gcc.gnu.org
11. MSYS, http://www.mingw.org/wiki/MSYS