INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

The United Nations University

SMR/774 - 10

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

CROSS-DEVELOPMENT OF EMBEDDED SYSTEMS

Chu Suan ANG
3 SS24/7
Taman Megah
47301 Petaling Jaya
MALAYSIA

Cross-Development of Embedded Systems

C S Ang

Cross-Development of Embedded Systems

Outline of Lectures

- Introduction to cross-development of embedded systems
  - What are embedded systems?
  - What is cross development?
  - Microcontroller/microprocessor resources used in embedded systems.
  - Designer's skills
  - Historical background of real-time embedded systems
  - Definition & classification of real-time embedded systems

- Design and development of embedded systems
  - System design
  - Choosing an embedded computer
  - Hardware design and development
  - Outline of hardware test procedure
  - Software design and development
  - State machine and state table
  - Simulators
  - Cross assemblers
  - Cross compilers
  - In-circuit Emulators and development systems

- A microcontroller (68HC811) embedded system
  - Architecture of the 68HC811
  - Hardware and software features.
  - Low-cost emulator
  - Cross-development in assembly language.
  - Cross-development in C.
  - A design example.

- A microprocessor (6809) embedded system
  - Architecture of the 6809.
  - Hardware features and software features.
  - Comparison of microcontroller with microprocessor.
  - Development in assembly language (resident & cross-assembly).
  - Cross-development in C.
  - A design example.

- A real-time kernel for embedded systems
  - Why real-time kernel?
  - uC/OS.
  - Kernel structure.
  - Interrupt processing.
  - Communication.
  - Examples.

What Are Embedded Systems?

- An embedded system is one with a built-in computer, typically for carrying out some kind of real-time applications. The computer in such a system does not behave as a stand-alone computing machine.

- There are numerous examples of embedded systems around us:
  - *Laboratory* - test equipment, data acquisition systems, control systems, dedicated equipment.
  - *Process industry* - process control system. This is the grand daddy of real-time embedded systems. Early examples are the closed-loop control system at a Texaco refinery in Texas in 1959 and a similar system at a Monsanto Chemical Company ammonia plant in Louisiana.
  - *Manufacturing industry* - production line assembly equipment, automatic test equipment, robots.
  - *Automotive* - engine controls, anti-lock braking, lamp, indicator and other controls.
  - *Consumer goods* - audio-visual equipment, microwave ovens, washing machines, dishwashers.
  - *Office & banking equipment* - autoteller, photocopiers, fax machines.
  - *Computer peripherals* - printers, keyboards, visual display units, modems.
  - *Aerospace* - flight management systems, engine controls.
  - *Telecommunications* - pagers, wireless phones, handphones.

- Although there are infinite varieties of embedded systems, the principles of operation, system components and design methodologies are essential the same. A typical system consists of a *computer* with its *standard input/output* devices and an *interface* to the physical environment, which may be a chemical plant, a car engine or a keyboard, for example.

- We shall deal with the development of such systems in general, with emphasis on a class of embedded systems using *microcontrollers* which is currently the most prevailing form of computer used in laboratory and many other applications.

What Is Cross Development?

- Developing application software for an embedded system requires development tools. When the development system and the *target system* have different types of CPU, the process is referred to as *cross development.*
  - For example, when a 486-based PC is used to develop software for a 68HC11 embedded system, it is cross development because the native compiler and assembler of PC which generate 486 codes cannot be used for the 68HC11. In this case cross-compiler and cross-assemler which runs in the PC but generates 68HC11 machine codes are needed.
- Cross development is necessary for a number of reasons:
  - Many microcontrollers used in embedded systems are just too small to be used as processors in development systems. Native or resident assemblers and compilers may not be available for such systems.
  - Existing computer facilities are readily available and with the appropriate cross-development software tools, are suitable for carrying out the task of software development. This is considered an important advantage because no extra hardware is needed and software tools such as editors are already available.
  - Some manufacturers are supporting their products with a dial-up facility or through Internet which allows users to download cross-assemblers and cross-compilers to the ubiquitous PC.
  - Everyone owns a PC or can lay hands on one `(quote Rinus Verkerk)`!

Microcontroller/Microprocessor Resources Used in Embedded Systems

- The evolution of microprocessor has been along two different paths. One of them has been the development of powerful CPU with 16- and 32-bit data bus and very large memory space (e.g. gigabytes). These processors are used in personal computers and workstations which form the backbone of computing facilities in home, commercial, educational, engineering and research environments.

- The power and speed of the 16- and 32-bit CPU of course do not limit them to the domain of stand-alone computers. They are used as embedded computers as wells. In fact they are used in many applications where sophisticated control or high speed operation is needed, e.g. HP Laserjet printers.

- However, it is true that for a large number of laboratory and other applications, the tasks can often be performed by a range of smaller processors - the *microcontrollers*. (Well, sledgehammer is not needed if it is a fly, nor should the butcher's knife be used when it is a chicken?)

- In this short series of lectures, we shall not deal with the development of embedded systems using 16- and 32-bit CPUs because of the complexities of such systems. However, their use as cross-development tools for microcontroller-based embedded systems will be elaborated.

- The second evolution path of microprocessor is along the line of microcontrollers which on a single chip the processor is integrated with RAM, ROM, EPROM, EEPROM, timers, serial and parallel I/O facilities. These microcontrollers are most suited for small real-time embedded systems or used as real-time modules in large systems.

Designer's Skills

- Good knowledge of the *microcontroller resources*. This should include the architecture, the instruction set, the addressing modes and the on-chip resources. The knowledge should generally extend beyond the simplified and idealised devices. For example, a good designer must know how the microcontroller handles interrupts and related timing issues so as to handle real-time activities effectively.

- Good knowledge of *real-time control*. The real-time requirement of the target system must be clearly understand before an effective solution may be found.

- Good knowledge of *software techniques* or software building blocks in handling various requirements and tasks of the target system.
  - For example, it may be an advantage to represent the system by a state machine. In this case, how can the state machine be implemented in software easily?
  - In an embedded system where a keyboard is used, how does one handle the keyboard parsing?

- Good knowledge of *hardware I/O components* or sub-modules. To be able to design a good embedded system, knowledge of the state-of-the-art peripheral devices is helpful.
  - For example, the technology of output devices including LED, LCD and CRT has progressed significantly. Manufacturers have implemented very sophisticated device drivers for some displays and it is a good idea to consider using them whenever possible.
  - Many embedded systems involve the use of ADC or DAC. Again, a good knowledge of accuracy, resolution, and speed of conversion is essential. If a target system is expected to measure 1 millidegree in 100 degrees, it is useless to design a system with a 10-bit ADC, for example.
  - Other components such as drivers, position control and position encoding are often used and should be included in the repertoire of hardware skill.

- Good knowledge of *development tools*. Development of embedded system requires both hardware and software development tools.
  - Hardware tools: multimeter, oscilloscope, logic probe, pulser, EPROM & EEPROM programmer, logic analyzer, in-circuit emulator, development system.
  - Software tools: editor, cross compiler, cross assembler and linker, simulator, development system.

Historical Background of Real-Time Embedded Systems

- Earliest proposal of using a computer in real-time application for controlling a plant:
  - Brown, G.S., Campbell, D.P., "Instrument engineering: its growth and promise in process-control problems', *Mechanical Engineering*, 72(2): 124 (1950).
- Early industrial installations of embedded systems:
  - September 1958 by Louisiana Power and Light Company for *plant monitoring* at a power station in Sterling, Louisiana.
  - First industrial *computer control* installation was by Texaco Company for a refinery at Port Arthur in Texas in March 1959.
- Many early systems were *supervisory control* systems that used steady-state optimisation calculations to determine the set points for standard analogue controllers.
- Later, *direct digital control* which allowed the direct control of plant actuators was added.
- The early real-time programs were written in *machine code* which was manageable when the tasks were well defined and the system small.
- In combining supervisory control with direct digital control the complexity of programming increased significantly. The two tasks have very different time scales and interrupting of the supervisory control is necessary. This led to the development of general purpose **real-time operation systems** and high-level languages for such systems.

Definition & Classification of Real-Time Embedded Systems

- Oxford Dictionary of Computing:
  - 'Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.'
- The above definition covers a wide range of systems:
  - from UNIX workstations
  - to aircraft engine control systems.
- An alternative definition:
  - A real-time system receives inputs and sends outputs to the target system at times determined by the target system operational considerations - not at times limited by the capabilities of the computer system.
- A real-time program is:
  - A program for which the correctness of operation depends both on the logical results of the computation and the time at which the results are produced.
- Classification of real-time systems:
  - Clock-based (cyclic, periodic) - process control systems.
  - Event-based (aperiodic) - alarm systems.
  - Interactive - autoteller, airlines reservation systems.
- Classification based on time constraints:
  - *Hard real-time* - must satisfy deadlines on each and every occasion, e.g. temperature controller of a critical process.
  - *Soft real-time* - occasional failure to meet deadlines acceptable, e.g. autotellers.

Design and Development of Embedded Systems

- System design.
- Design and build hardware.
- Design and develop software.
- Integrate software into target system.
- For very small projects involving only one person, the above tasks are carried out sequentially in that order. However, for bigger projects, it is often possible to develop the hardware and the software in parallel. This calls for a thorough system design in the first place.

System Design

- Define the functions and requirements of the target system. The *problem* must be well defined. Otherwise there is no *solution*. Difficulties arise when the scope of the work is not rigidly known or when the designer is uncertain of the capabilities of the various hardware and software resources. `(Well, talk to someone may help!)`
- Specify the *interface* to the target system clearly, for example:
  - Number and type of parallel I/O needed for interacting with the target system.
  - What kind of real-time requirement is needed?
  - Any serial communication needed? If so, what is the distance of communication?
  - Is the target system localised or distributed over a wide area?
  - Any ADC and DAC requirement? If so, what are the requirements on resolution, accuracy and sampling rate?
- Is it a networked or a stand-alone system? In the case of distributed or networked application, define the type of networking facility to use. This usually depends on the data rate and response time.
  - If the data rate requirement is kbps and below and the response time requirement is around a second, a low cost serial link based on RS232 or RS422 interfaces may be used.
  - If a high data rate up to Mbps is needed, use a standard LAN-type link, Ethernet or Token Ring for example.
- Specify the *user interface*. Is it an instrument panel-type interface? Or is it a *graphical user interface* (GUI)? Design a friendly user interface.

Choose An Embedded Computer

- Choose an appropriate microcontroller/microprocessor. The choice really depends on many factors, amongst them are:
  - Unique functional requirements of the target system. It may be that the ADC requirement calls for a particular processor, or the temporary buffer needed dictates another. Other applications may require a microcontroller with EEPROM as non-volatile storage.
  - Production volume of the target system. A one-off laboratory embedded system may use an expensive or oversized processor whereas a system that has to be produced in quantity may be very cost sensitive. One may have to use a $1 processor with masked ROM instead of $50 processor with EEPROM.
  - Experience of the designer. (Yes, one can learn a new processor easily these days, but won't it be simpler if one does not have to learn a new one at all?)
  - Availability of the devices. (There is no point having a design on paper only, other than for teaching purposes!?)
  - Your boss says 'use microcontroller xyz'.
- There is yet another alternative - obtain or purchase general purpose embedded computers with the necessary I/O and build only the interface to the *outside world*. This is an attractive option if you can afford it. There are manufacturers producing a wide variety of embedded computers ranging from 8-bit microcontroller-based systems to full-fledged 486 PC with 1.44MB ROM disk on a single expansion card!

Hardware Design and Development

- The beauty of designing embedded systems using microcontrollers and small microprocessors is the relative ease and simplicity. You no longer have to be a 20-year-experienced-electronic-engineer to be able to design the hardware. You may be a software programmer, a system analyst, a physicist or even a manager or director of college! (The folly is that you no longer can tell your boss that you cannot do it because you are not an electronic engineer?)

- Build, test and debug the hardware. Once the circuit design is completed, the next step is circuit board layout and fabrication. Unfortunately the hardware development process does not end there. In most cases, a certain degree of hardware testing and debugging must be done.

- To carry out these tasks, it would be advantageous if sophisticated tools such as development system, in- circuit emulator and logic analyser are available. However, it is possible to test and debug with the basic electronics laboratory equipment such as multimeter, oscilloscope and function generator alone, if a systematic approach is adopted.

Outline of Hardware Test Procedure

- Printed circuit board (PCB) inspection for track continuity and possible bridging. This is a step that is often overlooked. However, it is a vital step because easily locatable faults if left undetected, usually cause much more debugging efforts at a later stage.
- Power up the bare PCB and check voltages.
- If it is a microprocessor-based system, such as the 6809, or a microcontroller-based system operating in *expanded multiplexed* mode, test the address bus and (partially) the data and control bus on the *hardware kernel* which is the processor itself.
  - In the case of 6809, this is done by forcing a NOP ($12) on the data bus by pulling up D1 and D4 to 5V via resistors and grounding all other data lines. It causes the continuous execution of NOP for all memory locations. This in turn results in A0 toggling at half the system clock rate, A1 toggling at half the rate of A0 and so forth. The address bus can thus be checked easily with an oscilloscope. In this test, data bus and control bus are partially verified. This step is skipped if the system is single-chip, micocontroller-based.
  - If a logic analyser is not available, implement a tight loop program in the EPROM such as a branch-to-itself loop (LOOP BRA LOOP). For 6809, this consists of two bytes ($20 $FE) and takes three machine cycles to execute. A two-byte reset vector is also needed in the EPROM. The execution of this very short program can be followed cycle by cycle on an oscilloscope and thereby confirming the proper operation, at least partially, of the data and control bus.
- Test routines for I/O ports which have input switches and output indicators can be written and tested. Commonly used routines include incrementing the binary value of the output port at a slow rate for visual inspection, reading status of switches and sending it to the output port. This stage of testing serves to verify the operation of I/O ports and to provide users with function selection. Normally on power up the system is programmed to check the status of the input switches and jump to appropriate test routines or the main program.

- Small test routines for other components in the system are then implemented. This includes testing the serial link, the timers, ADC and the memories.
- In some embedded systems where the memory is not very small, a monitor program or kernel is then implemented.
- At this stage most of the hardware testing are done and the task moves on to application software testing and debugging. However, there is one type of hardware bug which is not detected by the testing mentioned above. These are problems caused by intermittent faults, glitches or external interference. These are detected by means of logic analyser or in-circuit emulator running in surveillance mode.

Software Design and Development

- The major task in software design is the breaking up of the entire application software into smaller manageable modules or components.
- The algorithm of the various processes in the system must be translated into the language of the embedded computer used.
- Finally, the integration of all the modules in a manner that can be tested and debugged easily is needed.
- The above tasks call for knowledge on software building blocks and software development tools.
- Unlike hardware development, the time taken in testing and debugging during software development can be surprisingly long if the design is not carried out systematically. (Well, one of the corollaries to Murphy's Law says that the time taken in software development can be computed by taking the initial estimate, multiplying it by 2 and changing the unit of time measurement to the next higher one. Thus, a project that you think will take you 2 weeks to complete will finally consume 4 months of your life. Ang's comment: it is an underestimate!)
- Fortunately, the facilities that support software development in the form of low-cost emulators, affordable simulators and cross-development software are now readily available. Many software tasks in embedded systems need not be implemented in assembly language anymore. It is now possible to develop most of the software of an embedded system in the comfort and power of high level languages like C.

State Machines and State Tables

- For a very small system, it is conceivable that the entire function of an embedded system be represented in a flowchart and implemented accordingly using a single program or a number of modules.
- There are however a number of shortcomings using the above method:
  - Testing of a monolithic program is often difficult.
  - Subsequent modifications of system function, like adding another control switch, are complex because the entire flowchart has to be revised and often re-implemented entirely.
- For many embedded systems, the complexities often justify a more systematic approach to designing the software. Representing the function of a system by a **state machine** is a very powerful method in developing an embedded system.
- The power of state machine representation comes from the fact that this state machine can subsequently be represented by a **state table** which is well suited for microcontroller and microprocessor implementation even at assembly language level.
- Using the state table method of implementing the functions of a system, it is natural that the task be broken down into small, more manageable and often independent modules, called the *action routines*. Such routines are more easily tested and often reusable.
- However, the single most important advantage of state table implementation really lies in the ease of function modification. In most cases, only the state table is modified together with the necessary new routines, while most of the old code would be intact.

- A simple example of a system with keyswitches and display.
  - Suppose we have a keypad with ten numeric keys 0 to 9 and two function keys ENTER and DELETE and a 4-digit numeric LED display.
  - On power up, the display shall show 0.
  - Numeric values can be entered on the keypad and as each digit is entered, it is scrolled into the display from the rightmost digit. During this mode, the display blinks to indicate digit entering mode.
  - The entering mode is terminated with either the ENTER key or the DELETE key.
  - If ENTER is pressed, the display stops blinking.
  - If DELETE is pressed, the display stops blinking and shows 0.
- There are 3 possible states in this example:

State	Name	Description
S0	Initial	Power-on state or after DELETE, display shows 0 in steady mode.
S1	Data Entry	Digit entry mode, display show digits in blinking mode.
S2	Display	Final display mode, display shows final value in steady mode

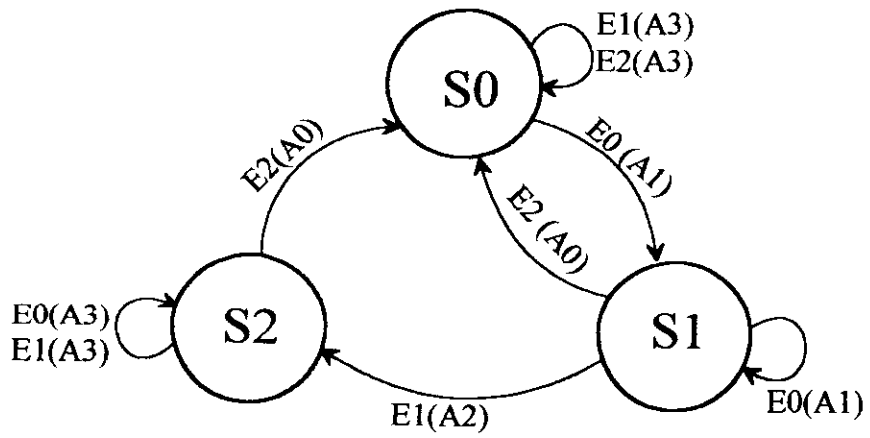
- There are 3 types of event:

Event	Name	Description
E0	Number	Entry of any numeric key.
E1	ENTER	ENTER key is pressed.
E2	DELETE	DELETE key is pressed.

- There are three action routines needed:

Action	Name	Description
A0	Reset	Display 0.
A1	Build digits	Build up display buffer from right while numbers are entered and blink display.
A2	Steady display	Show steady display.
A3	Null	No action.

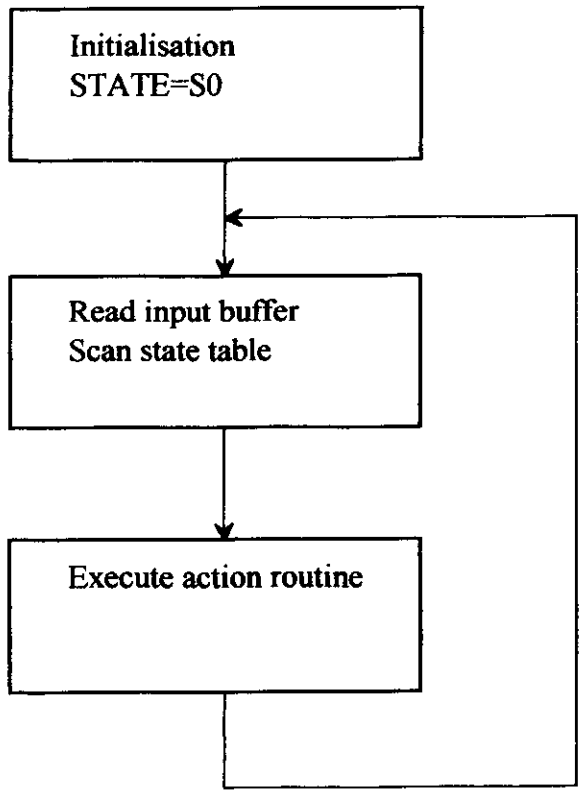
- State diagram.



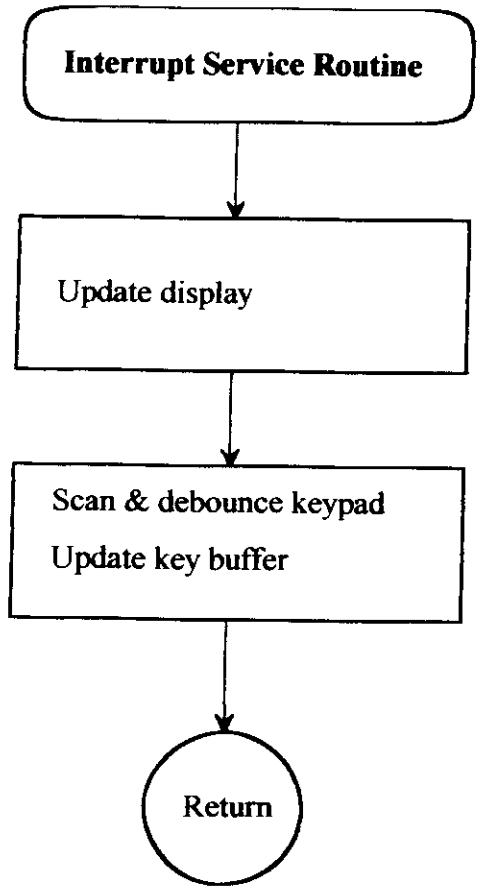
- State table.

Present State	Event	Action	Next State
S0	E0	A1	S1
	E1	A3	S0
	E2	A0	S1
S1	E0	A1	S1
	E1	A2	S2
	E2	A0	S0
S2	E0	A3	S2
	E1	A3	S2
	E2	A0	S0

- The complexity of the system has thus been broken down into:
  - A number of action routines.
  - A service routine to scan the keypad and update display.
  - A state stable.
  - A very small main program.
- The main program:



- The keypad and display service routine may be implemented as an interrupt service routine based on 10-ms clock ticks from a programmable timer module for example.



Simulators

- Simulators are programs that run on a computer, normally with a different processor from that of the target system, and *simulate* in software the operation of the target system.
- Many simulators are available in the PC environment now.
- The user interface of a simulator is usually a window environment showing simultaneously a number key information required in the tracing or debugging of a program.
- For example, the following windows are typically shown in the Assembly Language screen:
  - Program window
  - Register window
  - Memory window
  - Status window
- When C code is executed, the simulator may switch the window to a C screen which has:
  - C code window
  - Variable window
- A command menu is available for carrying many other functions. An example is as follows:
  - File menu - for loading & executing code file, assigning port file, etc.
  - Set - for setting breakpoints, selecting interrupt, etc.
  - Clear - for clearing breakpoints, clearing interrupts, etc.
  - Display - for displaying parameters, symbols, disassembled code, etc.
  - Modify - for changing window addresses, register contents, etc.
  - Go - for starting or resuming program execution.
  - Undo - for undoing a number of instructions.
  - Option - for setting options such as subroutine skipping, trace on/off.
  - Exit - for getting out of the simulator.

Cross Assemblers and Linkers

- Cross assemblers are programs that run on a computer with a different processor from that of the target system, and *assemble* programs written for the target system into *relocatable object code*.
- The linkers then relocate, usually with other object modules such as library modules, to the desired execution addresses for the target machine.
- Many cross assemblers for microprocessors and microcontrollers are available for the PC.
- Common features of cross assemblers are:
  - Provision for using macros in program, thus *macro-assembler*.
  - Conditional assembly.
  - Assembly time calculations.
  - Listing control.

Cross Compilers

- Cross compiler are programs that run on a computer with a different processor from that of the target system, and *compile* high level language programs written for the target system typically into assembly language programs.
- The use of cross compiler can reduce program development time significant for large project.
- It also make programs more portable, since they are written in high level languages such as C.
- A typical cross compiler consists of:
  - Macro preprocessor
  - Parser
  - Optimizer
  - Code generator

Cross Compilation from C to Executable Code

- The procedure for getting a program written in C to run in an embedded system consists of the following steps:
  1. Write a *run time startup routine* in assembly languge. `(Well, you can't escape, can you?)`
  2. Assemble the startup routine into object module.
  3. Write your application program in C.
  4. Compile your C program into assembly language program. There may be more than one program or file.
  5. Assemble your application program(s) into object module(s).
  6. Link the run time startup module, the application program module(s) and all necessary libray files to produce the run time module.
- **Run Time Startup Routine.** The function of this assembly language routine is to do the following:
  - Set up the stack. This is done before calling a C function. The stack stores function arguments, return addresses and local variables. It is also used by others including library routines and assembly language routines.
  - Call the C *main* fucntion.
  - Provide low level routines for I/O or other hardware related functions.
  - Move the constant and initialized data section into RAM if the program executes out of ROM.

An Example of Startup Routine

- The following is an example of a run time startup routine for a 6809 system based on the 2500AD C compiler.
- The default name for the startup routine is c6809r.src
- *Setting up stack pointer.*

```
stack:      .equal      7fffh              ;set stack below EPROM
c6809rt_startup:
            ldd         #stack
            subd        #lib_temp_constants_size
                                           ;allocate Lib temporary storage
            std         __lib_temp_ptr     ;Store ptr to temporary storage
            tfr         d,s
```

- *Defining sections.* The compiler uses four data classes:
  - **program**,
  - **const_data**,
  - **init_data** and
  - **uninit_data**.

  And the corresponding library sections:
  - **lib_program**,
  - **lib_const_data**,
  - **lib_init_data**.

  Plus two other sections:
  - **lib_temp_constants** and
  - **lib_temp_constant_end**.

- The linker links sections in the order they are defined. The run time startup routine must define all the sections even if some other empty. This allows the linker to calculate the size of constant and initialised data. An example:

```
;       Define sections that will be used.  Doing it here generates the
;       proper linking order.

vectors:                    .section    ;restart & interrupt vectors section
vectors_addr:               .equal      $
page0:                      .section    page0 ;page0 section
page0_addr:                 .equal      $
bank_table:                 .section    ;bank table section
bank_table_addr:            .equal      $
program:                    .section    ;program section
prog_addr:                  .equal      $
runtime_program:            .section    ;start of runtime support section
runtime_program_addr:       .equal      $
lib_program:                .section    ;library program section
lib_program_addr:           .equal      $
const_data:                 .section    ;constant data section
const_data_addr:            .equal      $
lib_const_data:             .section    ;library constant data
lib_const_data_addr:        .equal      $
const_data_end:             .section    ;end of program & constant data
const_data_end_addr:        .equal      $
init_data:                  .section    ;initialised data section
init_data_addr:             .equal      $
runtime_init_data:          .section    ;initialised data section
runtime_init_data_addr:     .equal      $
lib_init_data:              .section    ;library init data section
lib_init_data_addr:         .equal      $
init_data_end:              .section    ;end of all initialised data
init_data_end_addr:         .equal      $
uninit_data:                .section    ;uninitialised data section
uninit_data_addr:           .equal      $
uninit_data_end:            .section    ;end of all uninitialised data
uninit_data_end_addr:       .equal      $

;       The following sections are used to generate offsets for the library
;       temporary storage locations.  They are actually on the stack, with
;       each location referenced as an offset from the value in stored in
;       '__lib_temp_ptr'.
;

lib_temp_constants:         .section    offset 0, ref_only
lib_temp_constants_end:     .section    stacked,  ref_only
lib_temp_constants_size:    .equal      $       ;size of temporary constants
```

- *Moving initialised data into RAM.*

```
;       move constant & initialised data into ram

              ldd   #init_data_end_addr
              subd  #init_data_addr
              beq   ?no_init_data           ;skip move if size = 0
              ldx   #const_data_end_addr    ;load addr of init data
              ldy   #init_data_addr         ;ld runtime address
?init_loop:   pshs  a                       ;save high byte of data size
              lda   ,x+                     ;load source byte
              sta   ,y+                     ;store source byte
              puls  a                       ;restore high byte of data size
              subd  #1                      ;decrement byte count
              bne   ?init_loop
```

- *Zeroing the uninitialised data area.* This is required by C.

```
;       zero uninitialised data area

?no_init_data:
              ldd   #uninit_data_end_addr
              subd  #uninit_data_addr ;get size of uninit data
              beq   ?no_uninit_data         ;skip initialisation if = 0
              ldx   #uninit_data_addr
?uninit_loop:
              clr   ,x+                 ;zero uninitialised data area
              subd  #1                  ;decrement byte counter
              bne   ?uninit_loop
?no_uninit_data:
              .equal $
              clra                      ;enable interrupts
              tfr   a,ccr
              bsr   _main               ;execute program
              bra   c6809rt_startup
```

- *Handling I/O functions.* If the C library functions for input and output are used, the following must be provided:
  - input and output jump table and
  - device drivers.

```
;The locations containing input and output device number are __standard_in
;and __standard_out and they must be declared global.
;The jump table must be named __input_table and __output_table and declared
;global.

            .global     __standard_in
            .global     __standard_out
            .global     __output_table
            .global     __input_table

__standard_in:    .word 0             ;init to serial input port
__standard_out:   .word 0             ;init to serial output port

__output_table:   .word serial_out    ;address of serial port output routine

__input_table:    .word serial_in     ;address of serial port input routine

;Device drivers
serial_in:  bsr   get_character
            rts

serial_out: bsr   store_character
            rts
```

A C Program Example for Cross Compilation

- An example of a C program used to illustrate the procedure of cross compilation and simulation is shown below.

```
/*****************************************************************************/
/*    A Four Function Calculator                                             */
/*    calc.c                                                                 */
/*****************************************************************************/

      .asm
      .linklist                 ;update listing after the link
      .symbols                  ;include all symbols
      .endasm

#define  sprintf nf_int_isprintf     /* use char & int only sprintf       */
#define  sscanf  ns_int_isscanf      /* use char & int only sscanf        */

#include "c68c11sr.h"                /* include the special function regs */
#include "c68c11io.h"                /* include printf prototypes         */

#define  TX_INTERRUPT_ENABLE  0x80   /* transmitter interrupt enable bit  */
#define  RX_INTERRUPT_ENABLE  0x20   /* receiver interrupt enable bit     */
#define  RX_RECEIVER_ENABLE   0x04   /* receiver enable bit               */
#define  RX_READY             0x20   /* receiver ready bit mask           */

extern void line_in (char *);        /* line input routine                */

char upper_case (char);              /* 'upper_case' returns a character  */
char *next_token (char *);           /* 'next_token' returns a char ptr   */
int index;                           /* general purpose index             */
int operand1, operand2, result;      /* allocate operand #1, operand #2   */
                                     /* and result storage                */
char input_buffer[30];               /* allocate input buffer             */
char result_buffer[30];              /* allocate result buffer            */

main ()
{
char operation;                      /* storage for operation character   */
char *input_buffer_ptr;              /* pointer to input buffer           */

/* Disable the serial port receiver and interrupts. */

SCCR2 &= ~(RX_INTERRUPT_ENABLE | TX_INTERRUPT_ENABLE | RX_RECEIVER_ENABLE);

  while (1)
  {
  update_simulator_screen ();        /* update the simulator display */
    for (index = 0; index != sizeof (input_buffer); index++)
    {
    input_buffer[index] = ' ';
    }
  line_in (input_buffer);            /* input a line */
  input_buffer_ptr = input_buffer;   /* initialize ptr to start of buffer */
    if (upper_case (input_buffer[0]) == 'Q')
    {
    break;                           /* exit program on quit comment */
    }
```

```
/* Convert operand #1, increment input buffer ptr to start of next token */

    if (sscanf (input_buffer_ptr, "%d", &operand1) == 0)
    {
    sprintf (input_buffer, "Input Error\n");
    }
    else
    {
    input_buffer_ptr = next_token (input_buffer_ptr);

/* Skip spaces and tabs and store the operation */

      while (*input_buffer_ptr == ' ' || *input_buffer_ptr == '\t')
      {
      input_buffer_ptr++;
      }
   operation = *input_buffer_ptr++;
      while (*input_buffer_ptr == ' ' || *input_buffer_ptr == '\t')
      {
      input_buffer_ptr++;
      }

/* Convert operand #2, increment input buffer ptr to start of next token */

      if (sscanf (input_buffer_ptr, "%d", &operand2) == 0)
      {
      sprintf (input_buffer, "Input Error\n");
      }
      else
      {
      input_buffer_ptr = next_token (input_buffer_ptr);

/* Perform the operation */

       switch (operation)
       {
       case '+':
         result = operand1 + operand2;
       break;

       case '-':
         result = operand1 - operand2;
       break;

       case '*':
         result = operand1 * operand2;
       break;

       case '/':
         result = operand1 / operand2;
       break;

       default:
         sprintf (input_buffer, "Illegal Operation\n");
       break;
       }

/* Convert and display the result */

     input_buffer_ptr += sprintf (input_buffer_ptr, " = %d  ", result);
     }
```

```
    }
    for (index = 0; index != sizeof (result_buffer); index++)
    {
    result_buffer[index] = ' ';
    }
  strcpy (result_buffer, input_buffer);
  }
}

/* Input a line up to a carriage return. */

void line_in (buffer_ptr)
char *buffer_ptr;                    /* ptr to character storage buffer */
{
char c1;                             /* scratchpad character            */

SCCR2 |= RX_RECEIVER_ENABLE;         /* enable the receiver             */
  while ((SCSR & RX_READY) == 0)     /* wait for receiver to go ready   */
  {
  }
c1 = SCDR;                           /* input character                 */
  while (c1 != '\r')
  {
  *buffer_ptr++ = c1;                /* store character in buffer       */
    while ((SCSR & RX_READY) == 0)   /* wait for receiver to go ready   */
    {
    }
  c1 = SCDR;                         /* input character                 */
  }
SCCR2 &= ~RX_RECEIVER_ENABLE;        /* disable the receiver            */
*buffer_ptr++ = '\n';                /* terminate character string      */
*buffer_ptr = 0;
}

/* Find the start of the next token */

char *next_token (string_ptr)
char *string_ptr;
{
  if (*string_ptr == '-')
  {
  string_ptr++;                      /* skip minus sign */
  }
  while (*string_ptr >= '0' && *string_ptr <= '9')
  {
  string_ptr++;
  }
return (string_ptr);
}

/* Convert character to upper case */

char upper_case (character)
char character;
{
  if (character >= 'a' && character <= 'z')
  {
  character -= 0x20;
  }
return (character);
}
```

Procedure for Cross Compilation and Simulation

- Assemble run time startup routine:
  - `x6809 c6809r.src`
  - The cross assembler will assemble the run time startup routine and create an object module using the same name and the default extension `.obj`.
- Compile application program:
  - `c6809 calc.c`
  - The compiler compiles the C program into assembly program and then the cross assembler is invoked to assemble the assembly program into object module of 6809 with the extension `.OBJ`.
  - Various switch options are available for controlling the compilation and assembly process. For example, a -S switch tells the compiler to produce an assembly output file `.ASM`. The assembly listing (`calc.asm`) of the C program `calc.c` is appended at the end for reference.
- Link all the necessary modules:
  - `link calc.lnk`
  - The linker command file `.lnk` contains information necessary to link the various modules.
- Run the linked program on the simulator:
  - `s6809 calc`
  - The simulator loads code file `calc.s19`, the symbol file `calc.sym` and the debug control file `calc.dcf`.
  - It displays the assembly language screen and runs the startup routine until it reaches the C function main.
  - The simulator then switches to the C screen.

- Linker command file calc.lnk.

```
* this linker response file will link the c6809 calc and c6809sr
* 'vectors' gets linked at fff0h
fff0
* 'page0' gets linked at 0
0
* 'program' gets linked at 8000h
8000h
* 'bank_table' gets stacked on top of 'program'

* 'runtime_program' gets stacked on top of 'bank_table'

* 'lib_program' gets stacked on top of 'runtime_program'

* 'const_data' gets stacked on top of 'lib_program'

* 'lib_const_data' gets stacked on top of 'const_data'

* 'const_data_end' gets stacked on top of 'lib_const_data'

* 'init_data' gets linked indirectly at 100h
@100
* 'runtime_init_data' gets linked indirectly on to of 'init_data'
@
* 'lib_init_data' gets linked indirectly on to of 'runtime_init_data'
@
* 'init_data_end' gets linked indirectly on top of 'lib_init_data'
@
* 'uninit_data' gets stacked on top of 'init_data_end', reference only
-
* 'uninit_data_end' gets stacked on top of 'uninit_data', ref only
-
* next filename is calc.obj
calc
* now that all sections have been placed, the rest can be stacked.
* the easiest way to do this is with auto stack mode. that way you don't
* have to worry about how many sections there are.
;
* no more input filenames

* output filename is calc.s19
calc
* libraries are c6809c.lib, c6809s.lib & c6809m.lib
c6809c      include library c6809c.lib in the output file
c6809s      include library c6809s.lib in the output file
c6809m      include library c6809m.lib in the output file
```

- Simulator command file calc.scf

```
mi      ffe8, ffe9, Program                ; Modify Port I/O Limits
fi      c6809t.in, ffe9h, Binary           ; Assign Input Port File
fk      ffe8                               ; Redirect Input from Keyboard
mks     ffe7, f000, Program                ; Modify Stack Limits
sb      end, 0, Program                    ; Set Program Breakpoint
ra      input_buffer, Row : 4, Width : Maximum  ; Display Global Expression
ra      result_buffer, Row : 5, Width : Maximum ; Display Global Expression
si      IRQ, 2500                          ; Set IRQ Interrupt Delay
ou      on                                 ; Update C Screen On
```

- Simulator showing an assembly language screen:

```
  File  Set  Clear  Display  Modify  Go  Undo  Options  Exit
 Registers                          PC : 8007
  DP : 00  U : 0000  A : 00
 CCR : d4  S : ffd1  B : 00
                     D : 0000
 E F H I N Z V C     X : ffd1
 1 1 0 1 0 1 0 0     Y : 0000

 Status                             c6809rt_startup:  .equal  $
 Chip    Undo  Trace    Br Bkpt                        ldd     #STACK
 6809     Off   Off       Off                          subd    #lib_temp_constants_si
 Subroutine   Cycles     Int                          std     __lib_temp_ptr
    Step            0    None                         tfr     d,s

                                    ;       Set up the S6551 Serial Port chip.

                                                      lda     #00011010b
                                                      sta     serial_control

 Memory
 0000 :  00 00 00 03  32 00 ff d1  00 00 00 00  00 00 00 00  ....2...........
 0010 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
 0020 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
 0030 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
 0040 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
 0050 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
 0060 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
 0070 :  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

- After running the startup routine, simulator showing the C screen:

```
  File  Set  Clear  Display  Modify  Go  Undo  Options  Exit
 main ---------------------- Functions : Step   Loops : Step   Update : Off
main ()
{
char operation;                         /* storage for operation character   *
char *input_buffer_ptr;                 /* pointer to input buffer           *

  while (1)
  {
  update_simulator_screen ();           /* update the simulator display */
    for (index = 0; index != sizeof (input_buffer); index++)
    {
    input_buffer[index] = ' ';
    }
  line_in (input_buffer);               /* input a line */
  input_buffer_ptr = input_buffer;      /* initialize ptr to start of buffer *

 Local Variables ------------------------ Global Variables
```

- After executing the calculator program:

```
File  Set  Clear  Display  Modify  Go  Undo  Options  Exit
line_in ------------------- Functions : Step   Loops : Step   Update : On
void line_in (buffer_ptr)
char *buffer_ptr;                          /* ptr to character storage buffer */
{
char c1;                                   /* scratchpad character            */

enable_interrupts ();                      /* enable receiver interrupts      */
  while ((c1 = getchar ()) != '\n')
  {
  Enter Port Input Character :
*
*buffer_ptr = 0;
disable_interrupts ();                     /* disable receiver interrupts     */
}
```

```
Local Variables ------------------------ Global Variables
buffer_ptr = 0x0330  (32 - ' ')
c1 = 0

   input_buffer[0] = 32 32 32 32 32 32 32 32 32 32 32 32 32 32
   result_buffer[0] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ..................
```

- After entering 123 + 456:

```
File  Set  Clear  Display  Modify  Go  Undo  Options  Exit
line_in ------------------- Functions : Step   Loops : Step   Update : On
void line_in (buffer_ptr)
char *buffer_ptr;                          /* ptr to character storage buffer */
{
char c1;                                   /* scratchpad character            */

enable_interrupts ();                      /* enable receiver interrupts      */
  while ((c1 = getchar ()) != '\n')
  {
  Enter Port Input Character :
*
*buffer_ptr = 0;
disable_interrupts ();                     /* disable receiver interrupts     */
}
```

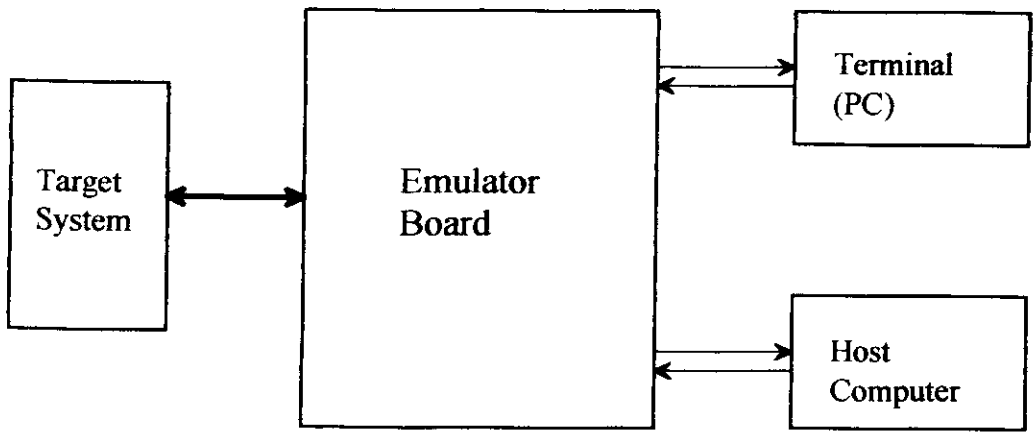
```
Local Variables ------------------------ Global Variables
buffer_ptr = 0x0330  (32 - ' ')
c1 = -78

   input_buffer[0] = 32 32 32 32 32 32 32 32 32 32 32 32 32 32
   result_buffer[0] = 49 50 51 43 52 53 54 32 61 32 53 55 57 32  123+456 = 579
```

Emulators

- Simulators are useful cross-development tools in many ways. There is however one aspect of system testing that simulators cannot do - to simulate *real-time* operation.
- Emulators are systems that overcome this shortcoming of simulators by accessing directly the target systems and running programs in real-time.
- They emulate the behaviour of the microcontroller or microprocessor in the target system and permit designers to load programs and to monitor and control the operation of the system.
- In the case of microprocessor-based systems, the target microprocessor is replaced by an emulating processor which has overall control over the data, address and control bus and thus the operation of the entire system.
- In the case of microcontroller-based systems, it is more complicated. Typically, the emulator operates the microcontroller in the expanded mode so as to gain access to the internal bus. It must also have:
  - extra RAM to hold the application software during development,
  - a monitor program, and
  - rebuilt ports to replace those lost in the expanded mode.
- Other features available in an emulator are:
  - communication facility between the monitor program and a host computer,
  - ability to download object code from the host computer to the target system,
  - ability to display and change RAM contents and processor status of the target system,
  - single stepping and breakpoint features, and
  - execution of the application program in full speed.

An Example of a Standalone Emulator

- A example of a low-cost standalone in-circuit emulator is the M68HC11EVM designed for developing 68HC11 embedded systems.
- It has the following features:
  - Emulate both the *single-chip* and *expanded-multiplexed* modes of operation.
  - Code may be generated using the resident assembler/disassembler, or may be downloaded through a host or terminal.
  - Microcontroller ROM is simulated by write-protected RAM during program execution.
  - Two serial links for host and terminal communication.
- The system operates in either one of two memory maps - the *monitor* map and the *user* map.
- Two types of memory map switching are possible. *Temporary* map switching allows modification of user memory, and *permanent* map switching allows execution of user programs.
- System connections.



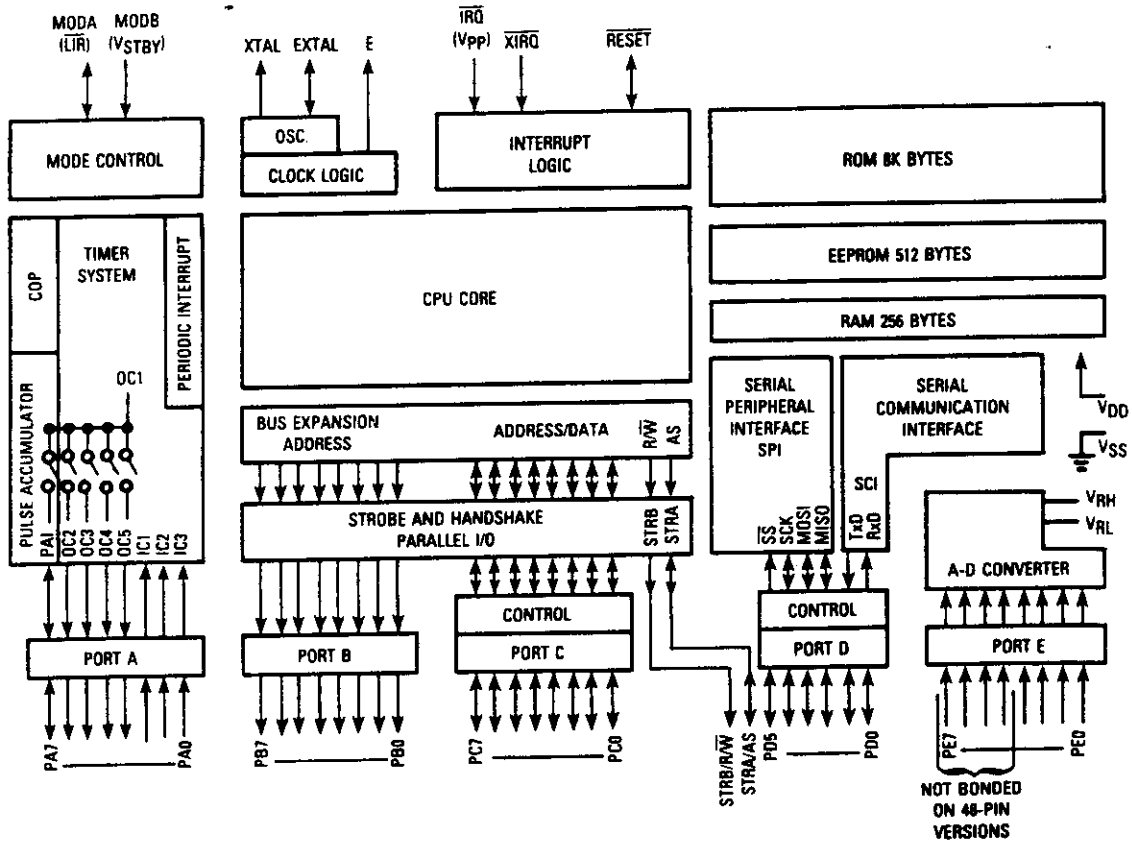
Development Systems

- Full-featured development systems are systems that support multiusers and multiple types of processor. An example is the HP64000.
- Basically, they provide editing, compilation, assembly, and emulation support.
- The design iteration cycle of *modify software*, *compile-assemble-link*, and *verify performance* is normally very well supported by user friendly and powerful software tools.
- Debugging is assisted by a *logic analyser* that monitors the performance of the target systems.
- Software performance analysis is also provided. This is a feature not available in low-cost development tools.
- `It costs $$$$$!`

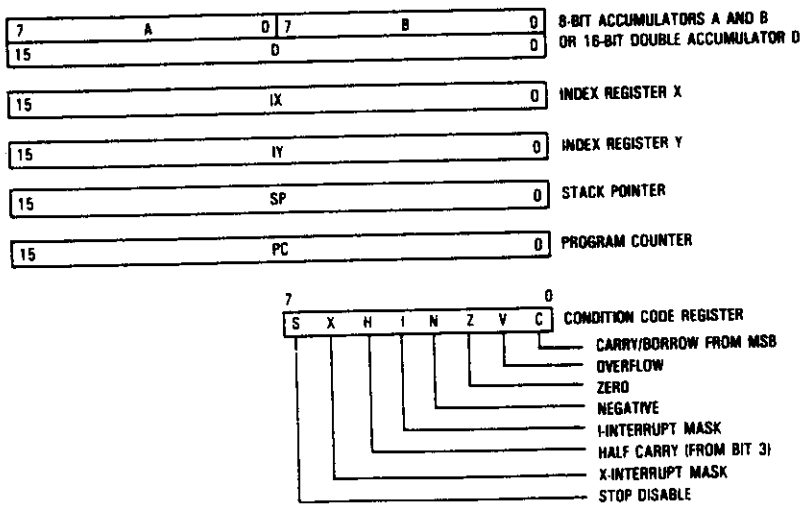
Microcontrollers

- As mentioned earlier, the inclusion of many I/O, memory, timer and communications resources on a single chip makes microcontrollers very useful in real-time embedded systems.
- Many semiconductor manufacturers are producing different families of microcontroller to suit various applications. The range is wide - from low cost (~$1) 4-bit chips to high performance 16-bit chips ($50~$100). The Intel 8096 is an example of a 16-bit microcontroller.
- However, the essential characteristics of the most of these microcontrollers are the same. We shall look at one - the Motorola 68HC11.
- The 68HC11 is a family of microcontrollers with members providing different I/O and memory facilities. They can be used in *single-chip* or *expanded mode*.
- The main features are:
  - *Parallel I/O* - 40 I/O lines arranged as five 8-bit ports, two general purpose and three fixed direction.
  - *ADC* - 8-channel, multiplexed-input, successive approximation with sample and hold. Conversion time 16 μs for 2 MHz system.
  - *Serial communications* - A full-duplex two-wire asynchronous serial communications interface (SCI) with baud rate ranges from 75 bps to 131 Kbps. A full-duplex three-wire synchronous serial peripheral interface (SPI) with a maximum master bit frequency of 1 MHz.
  - *Programmable Timer* - 16-bit with four stage prescaler, three capture functions and five output compare functions.
  - *Memories* - ROM (4K, 8K or 12K), EPROM (4K or 12K), EEPROM (512, 2K or 8K), RAM (256, 512 or 1K).
  - *Others* - 8-bit pulse accumulator, real-time interrupt and watchdog.

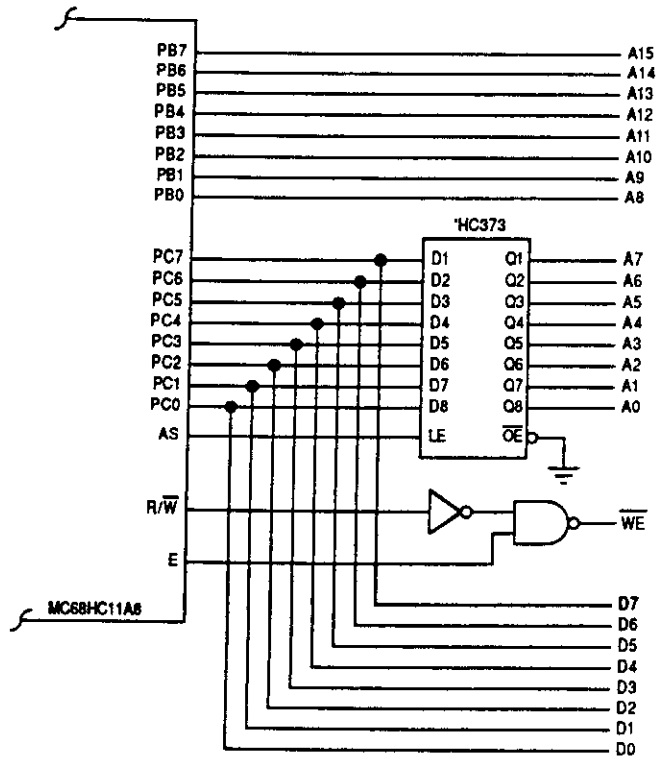
- Architecture of 68HC11.



- The programming model.
  - Enhanced M6800/M6801 instruction set (91 new opcodes).
  - 16-bit integer and fractional divisions.
  - Bit manipulation

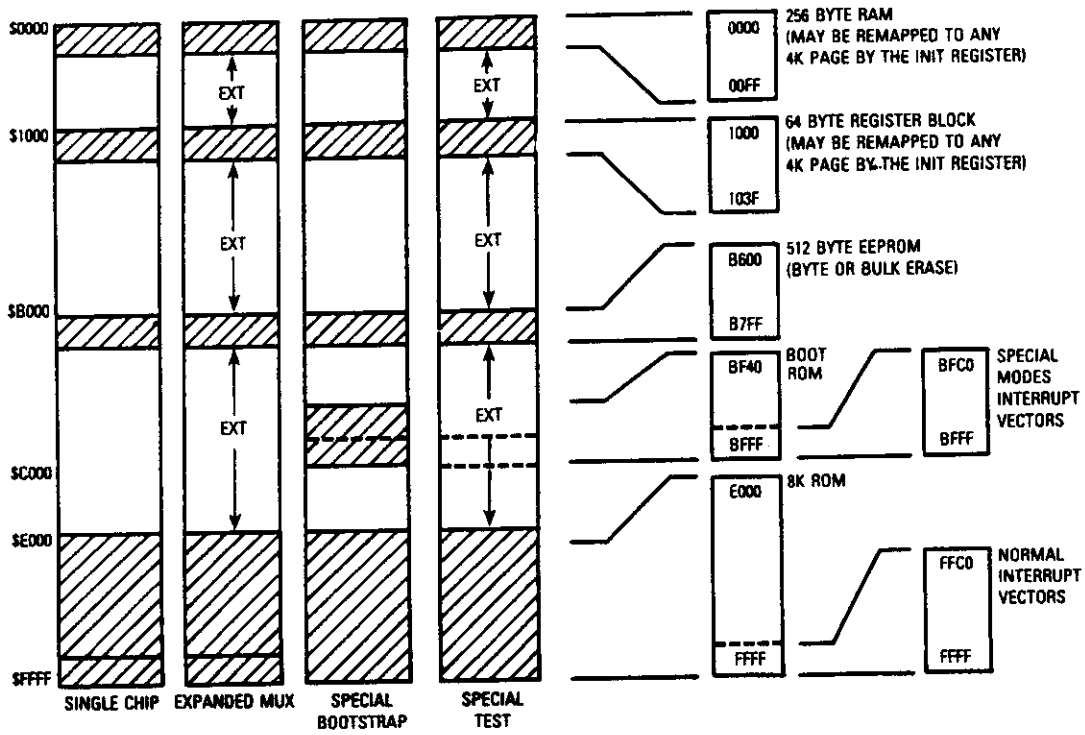


- Single-chip operating mode. The chip functions as a monolithic microcontroller without external address or data bus.
- Expanded-multiplexed operating mode. The chip can access a 64KB address space.
  - The total address space includes the on-chip memory addresses.
  - The expansion is made up of port B and port C, and control signals AS and R/W.



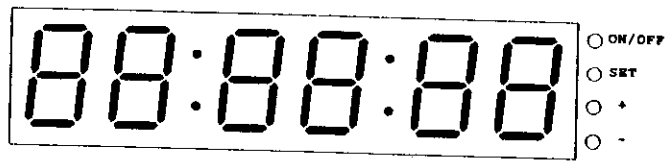
- Bootstrap operating mode. A special operating mode that uses a boot loader program in the bootstrap ROM to load program into RAM via SCI.
- Special test operating mode. This is a factory testing mode similar to the expanded-multiplexed mode except that the reset and interrupt vectors are fetched from external memory locations.

- Memory maps.
  - In expanded mode, the areas not used internally are for external memory and I/O.
  - If an external memory or I/O device is located to overlap an enabled internal resource, the internal resource will take priority.

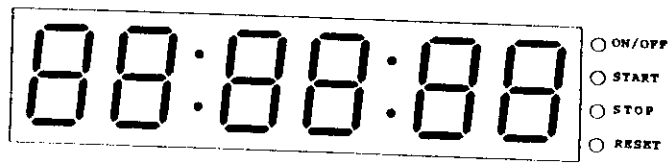


A Design Example Using 68HC11

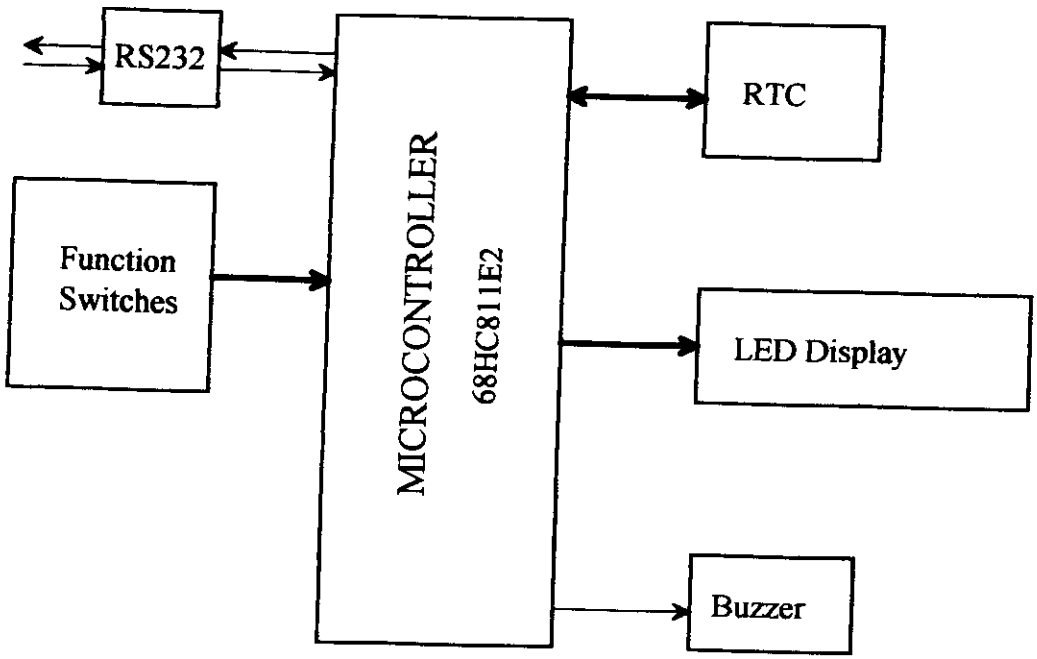
- An example of a small embedded system using a 68HC11 in the single-chip mode.
- The system functions as either a *Digital Clock* or a *Digital Timer*.
- *Digital Clock.*
  - Display time in 12 or 24-hour mode.
  - A backup battery is used to keep time in the real-time-clock chip in case of power failure.
  - Three push-button switches (**SET**, +, and –) are used to set time.



- *Digital Timer.*
  - Count-up timer in **HH:MM:SS**.
  - Three push-botton switches (**START**, **STOP** and **RESET**) for usual timer control.



- System configuration.
  - A MC68HC811E2 as the embedded microcontroller.
  - A MC146818A RTC chip with battery backup to provide real time.
  - An LED display submodule for displaying time.
  - A set of function selection switches.
  - A buzzer used as audio feedback.
  - An optional RS232 serial link for external setting of time.



- Software design. Although the operation of the system is relatively simple, it is still implemented as a number of small subroutines for ease of testing and upgrading.

#	Subroutine	Description
1	BOOTSTRAP	Initialise, read switches and jump to CLOCK, TIMER or test routines.
2	INITIAL	Initialse registers, I/O and RTC.
3	DISPLAY	Test display by cycling digits: "11:11:11", "22:22:22" ...
4	ALL	Test all functions by reading time from RTC and displaying it.
5	TIMER	Count up timer.
6	CLOCK	Real-time clock.
7	MIN_INC	Increment minute_field in time_buffer.
8	HOUR_INC	Increment hour_field in time_buffer.
9	TEST_SECOND	Detect closure of button_1 for synchronisation.
10	NO_KEY	Detect release of all buttons.
11	BZR	Beep.
12	DELAY	Delay.
13	S_DELAY	Short delay.
14	BZ_DELAY	Beeping duration.
15	WATCHDOG	Arm watchdog.
16	UPDATE_DISP	Update display_buffer, send it to display module.
17	SET_RTC	Set RTC.
18	UPDATE_RTC	Update RTC.
19	UPDATE_TIME	Update time_buffer.
20	WRITE_RTC	Write one byte into RTC RAM.
21	READ_RTC	Read one byte from RTC RAM.
22	RTC_IRQ	Interrupt service routine requested by RTC at 4HZ.

Appendix I Assembly Listing of cal.c

```
;calc.asm

;An assembly program produced by c6809 cross compiler.  The C source
;program is calc.c

program:     .section
const_data:  .section
init_data:   .section
uninit_data: .section
bank_table:  .section
             .program
             .llchar     ?
             .longchk    off
             .extern     page0 __oper1
             .extern     page0 __oper1_high8
             .extern     page0 __oper2
             .extern     page0 __oper2_high8
             .extern     page0 __lib_temp_ptr
;.asm
;
             .linklist                ;update listing after the link
             .symbols                 ;include all symbols
;.endasm
;
;int  index;
;
             .uninit_data
             .i_align
_index:      .ds   2
;int operand1= operand2= result;
;
;
             .i_align
_operand1:   .ds   2
             .i_align
_operand2:   .ds   2
             .i_align
_result:     .ds   2
;char input_buffer[30];
;
_input_buffer:     .ds   30*1
;char result_buffer[30];
;
;
_result_buffer:    .ds   30*1
;
             .program
_main:             .equal      $
;
;
?ASC0:             .equal      0
?TSC0:             .equal      2
?LSC0:             .equal      3
             leas  -?TSC0-?LSC0,s
;while
?TOL0:             .equal      $
;(1)
;
```

```
                lda   #1
                cmpa  #0
                lbeq  ?BOL0
 ?PLC0:         .equal      $
 ;update_simulator_screen ()
                jsr   _update_simulator_screen
 ;index = 0;
                clr   _index
                clr   _index+1
 ?FLC1:         .equal      $
 ;index != sizeof (input_buffer);
                ldd   _index
                cmpd  #30
                lbeq  ?BOL1
                jmp   ?FLB1
 ?FLA1:         .equal      $
 ;index++)
 ;
                ldd   _index
                addd  #1
                std   _index
                jmp   ?FLC1
 ?FLB1:         .equal      $
 ;input_buffer[index] = ' '
                ldd   _index
                addd  #_input_buffer
                std   __oper2
                lda   #32
                sta   [__oper2]
 ;
                jmp   ?FLA1
 ?BOL1:         .equal      $
 ;line_in (input_buffer)
                ldd   #_input_buffer
                jsr   _line_in
 ;input_buffer_ptr = input_buffer
                ldd   #_input_buffer
                std   -3+?LSC0,s
 ;(upper_case (input_buffer[0]) == 'Q')
 ;
                lda   _input_buffer+0
                jsr   _upper_case
                cmpa  #81
                lbne  ?BOL2
 ?PLC1:         .equal      $
 ;break;
 ;
                jmp   ?BOL0
 ;
 ;(ns_int_isscanf        (input_buffer_ptr, "%d", &operand1) == 0)
 ;
 ?BOL2:         .equal      $
                .const_data
 ?PLC2:         .byte "%d",0
                .program
                ldd   #_operand1
                pshs  d
                ldx   #?PLC2
                ldd   -3+?LSC0+2,s
                jsr   _ns_int_isscanf
                leas  2,s
                cmpd  #0
```

```
            lbne  ?BOL3
?PLC3:            .equal      $
;nf_int_isprintf   (input_buffer, "Input Error\n")
            .const_data
?PLC4:            .byte "Input Error",0ah,0
            .program
            ldx   #?PLC4
            ldd   #_input_buffer
            jsr   _nf_int_isprintf
;
;else
;
            jmp   ?EOI3
?BOL3:            .equal      $
;input_buffer_ptr = next_token (input_buffer_ptr)
            ldd   -3+?LSC0,s
            jsr   _next_token
            std   -3+?LSC0,s
;while
?TOL4:            .equal      $
;(*input_buffer_ptr == ' ' || *input_buffer_ptr == '\t')
;
            lda   [-3+?LSC0,s]
            cmpa  #32
            lbeq  ?PLC5
?OCC0:            .equal      $
            lda   [-3+?LSC0,s]
            cmpa  #9
            lbne  ?BOL4
?PLC5:            .equal      $
;input_buffer_ptr++
            ldd   -3+?LSC0,s
            addd  #1
            std   -3+?LSC0,s
;
            jmp   ?TOL4
?BOL4:            .equal      $
;operation = *input_buffer_ptr++
            ldd   -3+?LSC0,s
            std   +0+?LSC0,s
            addd  #1
            std   -3+?LSC0,s
            ldd   +0+?LSC0,s
            tfr   d,y
            lda   ,y
            sta   -1+?LSC0,s
;while
?TOL5:            .equal      $
;(*input_buffer_ptr == ' ' || *input_buffer_ptr == '\t')
;
            lda   [-3+?LSC0,s]
            cmpa  #32
            lbeq  ?PLC6
?OCC1:            .equal      $
            lda   [-3+?LSC0,s]
            cmpa  #9
            lbne  ?BOL5
?PLC6:            .equal      $
;input_buffer_ptr++
            ldd   -3+?LSC0,s
            addd  #1
            std   -3+?LSC0,s
```

```
;
                jmp     ?TOL5
?BOL5:          .equal      $
;(ns_int_isscanf          (input_buffer_ptr, "%d", &operand2) == 0)
;
                .const_data
?PLC7:          .byte "%d",0
                .program
                ldd     #_operand2
                pshs    d
                ldx     #?PLC7
                ldd     -3+?LSC0+2,s
                jsr     _ns_int_isscanf
                leas    2,s
                cmpd    #0
                lbne    ?BOL6
?PLC8:          .equal      $
;nf_int_isprintf   (input_buffer, "Input Error\n")
                .const_data
?PLC9:          .byte "Input Error",0ah,0
                .program
                ldx     #?PLC9
                ldd     #_input_buffer
                jsr     _nf_int_isprintf
;
;else
;
                jmp     ?EOI6
?BOL6:          .equal      $
;input_buffer_ptr = next_token (input_buffer_ptr)
                ldd     -3+?LSC0,s
                jsr     _next_token
                std     -3+?LSC0,s
;(operation)
;
                ldb     -1+?LSC0,s
                sex
                std     __oper2
?PLC10:         .equal      $
;
                ldd     #43
                subd    __oper2
                lbeq    ?SWL0
                ldd     #45
                subd    __oper2
                lbeq    ?SWL1
                ldd     #42
                subd    __oper2
                lbeq    ?SWL2
                ldd     #47
                subd    __oper2
                lbeq    ?SWL3
                lbra    ?SWL4
;case '+':
;
?SWL0:          .equal      $
;result = operand1 + operand2
                ldd     _operand1
                addd    _operand2
                std     _result
;break;
;
```

```
;
            jmp   ?BOL7
;case '-':
;
?SWL1:              .equal      $
;result = operand1 - operand2
            ldd   _operand1
            subd  _operand2
            std   _result
;break;
;
;
            jmp   ?BOL7
;case '*':
;
?SWL2:              .equal      $
;result = operand1 * operand2
            ldd   _operand2
            std   __oper2
            ldd   _operand1
            jsr   __mult_int
            std   _result
;break;
;
;
            jmp   ?BOL7
;case '/':
;
?SWL3:              .equal      $
;result = operand1 / operand2
            ldd   _operand2
            std   __oper2
            ldd   _operand1
            jsr   __div_int
            std   _result
;break;
;
;
            jmp   ?BOL7
;default:
;
?SWL4:              .equal      $
;nf_int_isprintf   (input_buffer, "Illegal Operation\n")
            .const_data
?PLC11:             .byte "Illegal Operation",0ah,0
            .program
            ldx   #?PLC11
            ldd   #_input_buffer
            jsr   _nf_int_isprintf
;break;
;
            jmp   ?BOL7
;
?BOL7:              .equal      $
;
;input_buffer_ptr += nf_int_isprintf       (input_buffer_ptr, " = %d  ",
result)
            .const_data
?PLC12:             .byte " = %d  ",0
            .program
            ldd   _result
            pshs  d
```

```
                ldx   #?PLC12
                ldd   -3+?LSC0+2,s
                jsr   _nf_int_isprintf
                leas  2,s
                std   __oper2
                ldd   -3+?LSC0,s
                addd  __oper2
                std   -3+?LSC0,s
    ;
    ?EOI6:              .equal      $
    ;
    ?EOI3:              .equal      $
    ;index = 0;
                clr   _index
                clr   _index+1
    ?FLC8:              .equal      $
    ;index != sizeof (result_buffer);
                ldd   _index
                cmpd  #30
                lbeq  ?BOL8
                jmp   ?FLB8
    ?FLA8:              .equal      $
    ;index++)
    ;
                ldd   _index
                addd  #1
                std   _index
                jmp   ?FLC8
    ?FLB8:              .equal      $
    ;result_buffer[index] = ' '
                ldd   _index
                addd  #_result_buffer
                std   __oper2
                lda   #32
                sta   [__oper2]
    ;
                jmp   ?FLA8
    ?BOL8:              .equal      $
    ;strcpy (result_buffer, input_buffer)
                ldx   #_input_buffer
                ldd   #_result_buffer
                jsr   _strcpy
    ;
                jmp   ?TOL0
    ?BOL0:              .equal      $
    ;
    ?BOF0:              .equal      $
                leas  ?LSC0+?TSC0+0,s
                rts
    ;
    _line_in:   .equal      $
                pshs  d
    ;
    ;
    ?ASC1:              .equal      0
    ?TSC1:              .equal      2
    ?LSC1:              .equal      1
                leas  -?TSC1-?LSC1,s
    ;enable_interrupts ()
                jsr   _enable_interrupts
    ;while
    ?TOL9:              .equal      $
```

```
;((c1 = getchar ()) != '\n')
;
            jsr   _getchar
            sta   -1+?LSC1,s
            cmpa  #10
            lbeq  ?BOL9
?PLC13:             .equal      $
;*buffer_ptr++ = c1
            ldd   +0+?LSC1+?TSC1,s
            std   +0+?LSC1,s
            addd  #1
            std   +0+?LSC1+?TSC1,s
            ldd   +0+?LSC1,s
            std   __oper2
            lda   -1+?LSC1,s
            sta   [__oper2]
;
            jmp   ?TOL9
?BOL9:              .equal      $
;*buffer_ptr++ = '\n'
            ldd   +0+?LSC1+?TSC1,s
            std   +0+?LSC1,s
            addd  #1
            std   +0+?LSC1+?TSC1,s
            ldd   +0+?LSC1,s
            std   __oper2
            lda   #10
            sta   [__oper2]
;*buffer_ptr = 0
            ldd   +0+?LSC1+?TSC1,s
            std   __oper2
            lda   #0
            sta   [__oper2]
;disable_interrupts ()
            jsr   _disable_interrupts
;
?BOF1:              .equal      $
            leas  ?LSC1+?TSC1+2,s
            rts
;
_next_token:        .equal      $
            pshs  d
;
;
?ASC2:              .equal      0
?TSC2:              .equal      0
?LSC2:              .equal      0
;(*string_ptr == '-')
;
            lda   [+0+?LSC2+?TSC2,s]
            cmpa  #45
            lbne  ?BOL10
?PLC14:             .equal      $
;string_ptr++
            ldd   +0+?LSC2+?TSC2,s
            addd  #1
            std   +0+?LSC2+?TSC2,s
;
;while
?BOL10:             .equal      $
?TOL11:             .equal      $
;(*string_ptr >= '0' && *string_ptr <= '9')
```

```
;
                lda     [+0+?LSC2+?TSC2,s]
                cmpa    #48
                lblt    ?BOL11
?ACC2:                  .equal      $
                lda     [+0+?LSC2+?TSC2,s]
                cmpa    #57
                lbgt    ?BOL11
?PLC15:                 .equal      $
;string_ptr++
                ldd     +0+?LSC2+?TSC2,s
                addd    #1
                std     +0+?LSC2+?TSC2,s
;
                jmp     ?TOL11
?BOL11:                 .equal      $
;return (string_ptr)
                ldd     +0+?LSC2+?TSC2,s
;
?BOF2:                  .equal      $
                leas    ?LSC2+?TSC2+2,s
                rts
;
_upper_case:            .equal      $
                pshs    a
;
;
?ASC3:                  .equal      0
?TSC3:                  .equal      0
?LSC3:                  .equal      0
;(character >= 'a' && character <= 'z')
;
                lda     +0+?LSC3+?TSC3,s
                cmpa    #97
                lblt    ?BOL12
?ACC3:                  .equal      $
                lda     +0+?LSC3+?TSC3,s
                cmpa    #122
                lbgt    ?BOL12
?PLC16:                 .equal      $
;character -= 0x20
                lda     +0+?LSC3+?TSC3,s
                suba    #32
                sta     +0+?LSC3+?TSC3,s
;
;return (character)
?BOL12:                 .equal      $
                lda     +0+?LSC3+?TSC3,s
;
?BOF3:                  .equal      $
                leas    ?LSC3+?TSC3+1,s
                rts
;
_update_simulator_screen:       .equal      $
;
;
?ASC4:                  .equal      0
?TSC4:                  .equal      0
?LSC4:                  .equal      0
;
?BOF4:                  .equal      $
                rts
```

```
                .global     _index
                .global     _result_buffer
                .global     _line_in
                .extern     _disable_interrupts
                .extern     _nf_int_isprintf
                .extern     _ns_int_isscanf
                .global     _upper_case
                .global     _result
                .extern     _getchar
                .extern     _strcpy
                .global     _input_buffer
                .global     _update_simulator_screen
                .global     _main
                .global     _next_token
                .global     _operand1
                .global     _operand2
                .extern     _enable_interrupts
                .extern     __mult_int
                .extern     __div_int
;
                .end
```