5 DOCUMENTATIONS TECHNIQUES

5.1 Noyau temps réel NTR++

La carte Lx 162 est équipée d'un noyau temps réel objet multi-threads romable NTR++. Les classes Thread, Semaphore et Event sont utilisables. On trouvera ci-dessous le fichier d'entête correspondant ainsi que la description UML.

5.1.1 les Threads

```
/*****************************************************************************
*       Module Name:            Thread.h
*****************************************************************************/
#ifndef _THREAD_H
#define _THREAD_H

/*-- Class Thread is the basic class for NTR++ thread control.
Usage example:
#include <Thread.h>

void threadEntryPoint( ) ;
void main( )
{ ....
 Thread first(threadEntryPoint);
 Thread second(threadEntryPoint, 50);
 ....
 first.cancel( ) ;
 second.cancel( ) ;
}
--*/
class Thread {          // The scheduler is based on the highest-priority ready task determined by two things :
                        // 1) the priority level you assign to the task when it is created
                        // 2) the order tasks are made ready among equal-priority tasks
public:         // create and start a thread with   1(low) =<priority =<255(high)   ( default=10)
        Thread (void (*startRoutine) ( ), int priority = 10);

        ~Thread ( );                    // delete the thread

        void    suspend ( );            // suspend thread

        void    resume ( );             // resume thread

        void    cancel ( );             // cancel thread

        static void     terminate ( );          // terminate current thread

        static void     sleep (long sec);       // delay current thread

        static void     tickSleep (long ticks); // sleep in timer tick units (1/100e s)

protected:
        int     pri;            // thread priority
        int     idThread;       // thread id
};
#endif /* _THREAD_H */
```

Thread
pri : int # idThread : int
+ Thread(startRoutine : void (*) (void), priority : int = 10) : Thread + ~Thread () + suspend() : void + resume() : void + cancel() : void + terminate() : void + sleep(sec : long) : void + tickSleep(ticks : long) : void

5.1.2 Les Sémaphores

```
/*****************************************************************************
*       Module Name:            Semaphore.h
*
*****************************************************************************
*/

#ifndef _SEMAPHORE_H
#define _SEMAPHORE_H

/*--
 Class Semaphore is interface to the NTR++ semaphore object.

 Usage example:

 #include <Semaphore.h>

 Semaphore s; // binary semaphore

 s.wait();
 // critical section code here
 s.post();

 Semaphore sem(2); // general semaphore,
          // can synchronize access to two resource units

 sem.wait();    // get one of the two resource units
 // utilize the resource
 sem.post();
--*/
class Semaphore {

public:
 Semaphore (int init_value = 1);

 ~Semaphore ();

 void post ();              // increment semaphore  V(s)

 void wait ();              // decrement semaphore (with blocking if sem=0) P(s);

 int trywait ();            // if sem >0 decrement semaphore P(s) and return 0
                            // if sem =0 return ER_NMP without blocking

 int getvalue ();           // return semaphore value

protected:
 int idSem;                 // semaphore id
private:
 int value;                 // semaphore value
};

#endif /* _SEMAPHORE_H */
```

Semaphore
idSem : int
- value : int
+ Semaphore(init_value : int = 1) : Semaphore
+ ~Semaphore()
+ post() : void
+ wait() : void
+ trywait() : int
+ getvalue() : int

5.1.3 les Evénements

```
/*****************************************************************************
*
*       Module Name:            Event.h
*
*****************************************************************************
*/

#ifndef _EVENT_H
#define _EVENT_H

/*-- Event class is interface to the NTR++ events
Usage example:

#include <Event.h>

Event  e1, e2;

void codeThreadOne( )          void codeThreadTwo( )
{.....                         {.....
 e1.set( );                     e1.wait( ) ;
  ...                           e1.clear( ) ;
e2.wait( ) ;                    ...
e2.clear( ) ;                   e2.set( );
....                            ....
}                               }

 --*/

class Event {

  public:
        Event ();

        ~Event ();

         void   set ();          // set event

        void    clear ();        // clear event

        void    wait ();         // wait event

        int     read ();         // return state of event

  protected:

        int idEvent;

};

#endif /* _EVENT_H */
```

Event
idEvent : int
+ Event() : Event + ~Event() + set() : void + clear() : void + wait() : void + read() : int

5.1.4 Exemple d'utilisation des objets NTR++

```
/*****************************************************************************
*       Module Name:            exempleBidon.cpp
*****************************************************************************/
#include <Thread.h>
#include <Event.h>
#include <Semaphore.h>
////////////////////////////////////////////////////////////////////////
// Definition d'une classe chantier
////////////////////////////////////////////////////////////////////////
class Chantier {
        Semaphore brouette(2);
        Event auSecours, secourir;
        Thread *chef;
 public:
//---------- Constructeur de chantier ----------
        Chantier ( )            { chef = new Thread(travailDeChef);}
//---------- Destructeur de chantier ----------
        ~Chantier( )            { delete chef;}
//---------- Travail d'un manoeuvre ----------
        void travailManoeuvre( ) {
          while (1) {
                brouette.wait( );               // prendre une brouette disponible
                for (int nbFois=0; nbFois <10; nbFois ++) {
                        .....           // remplir la brouette de sable ...
                                if (KO) { auSecours.set( );     // appel au secours
                                          secourir.wait( );     // attendre secours
                                          secourir.clear( );
                                        }
                        }
                brouette.post( );               // libérer la brouette
                Thread::sleep(10*60);           // repos de 10 mn bien mérite
                }
        }
//---------- Travail d'un secouriste ----------
void travailPompier( ) {
        while (1) {
                auSecours.wait( );              // attente d'un appel
                auSecours.clear( );             // acquitter appel
                secourir.set( );                // ranimer victime
                }
        }

//---------- Travail du Chef ----------
void travailDeChef( ) {
        Thread victor(travailPompier);          //embauche d'un secouriste
                {// Creation de 3 Threads
                Thread albert(travailManoeuvre);        // embauche de 3 ouvriers
                Thread julien(travaiManoeuvre);
                Thread emile(travailManoeuvre);
                Thread::sleep(8*3600);                  // le chef dort toute la journee
                } // fin de bloc : les 3 Threads ouvriers de classe automatique meurent.
        victor.cancel( ) ;      // mort de victor
        Thread : : terminate( );        // le chef se suicide !
        }
}; // fin Chantier

// Exemple d'utilisation de la classe Chantier
void main (void) {
Chantier maMaison;
}
```

5.2 Le protocole HTTP 1.0

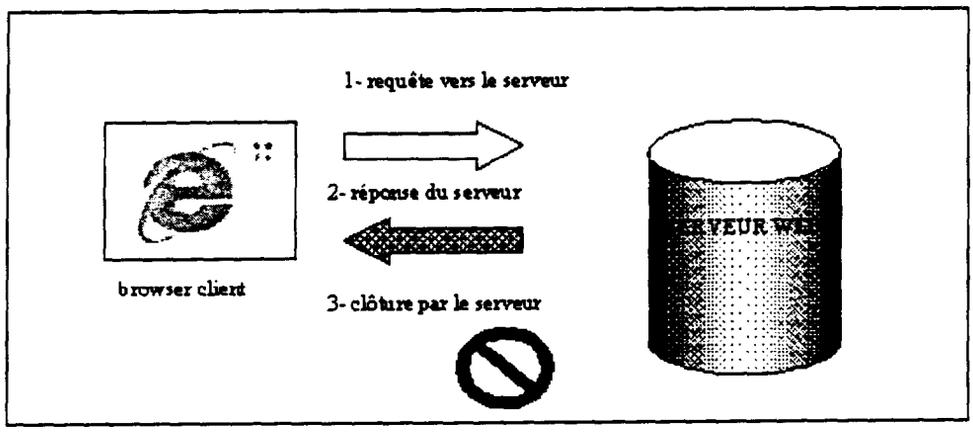
L'HyperText Transfer Protocol (HTTP) est un protocole de niveau application, léger et efficace, pour la transmission de documents distribués et multimédia. Le protocole HTTP 1.0 est décrit dans la RFC 1945. Le résumé ci-dessous est une libre inspiration de différents documents disponibles sur le réseau Internet. Il est forcément incomplet et peut comporter certaines imprécisions.

5.2.1 Présentation

Ce protocole fixe le principe de communication entre un logiciel de navigation Web (communément appelé "navigateur" ou « browser » ou « client ») et un serveur Web (souvent appelé httpd , d pour daemon sous Unix).

Le principe de communication est simple :

- La communication s'initie TOUJOURS à la demande du client. Celui ci s'adresse à un serveur caractérisé par une adresse IP ou son nom, sur un port connu, le plus souvent le port 80, mais cela n'a rien d'obligatoire.
- Une connexion TCP s'établit.
- Le client envoie une *requête* au serveur et ce dernier lui renvoie une *réponse* correspondant à sa requête.
- Le serveur clôt la connexion une fois la totalité de la réponse émise.



Nous allons maintenant détailler une requête et une réponse HTTP 1.0.

5.2.2 Requête HTTP

Le format d'une requête HTTP est le suivant :

Ligne de commande
En-tête de la requête
[ligne vide]
Corps de la requête

- La ***ligne de commande*** possède elle même trois champs : *commande, URL et version.*

  - Le premier champ, *commande*, contient une des commandes définies dans le protocole HTTP. Les principales commandes sont les suivantes :
    - GET : demande au serveur de renvoyer le contenu de l'information pointée par l'URL spécifiée dans la ligne de commande. Il peut s'agir d'un simple fichier HTML ou multimédia (image, son, ...), voire le compte rendu de l'exécution d'un programme CGI.
    - HEAD : cette commande est similaire à la précédente mais le serveur ne renvoie que l'en-tête associé à la ressource demandée (par exemple, la date de dernière modification d'un fichier, ...).

- POST : permet au client d'envoyer des données au serveur, comme par exemple le contenu d'un formulaire renseigné par l'utilisateur. Ces données constituent le *corps de la requête.*

- Le deuxième champ de la ligne de commande est une *URL*. Elle désigne en fait la ressource sur laquelle on désire appliquer la commande spécifiée dans le champ précédent. Comme nous l'avons dit à l'instant, cette URL peut aussi bien désigner un fichier statique (HTML, son, ...) ou un programme CGI (Cf. chapitre suivant). Une URL bien formée ressemble à ceci: http:// host[:port]/chemin/file.ext
  Le numéro de port par défaut est 80.

- Le dernier champ, *version*, contient la version du protocole HTTP implémenté dans le client considéré. La syntaxe est la suivante : HTTP/*version*. Exemple : HTTP/1.0

♦ Etudions maintenant l'***en-tête*** associé à la requête exprimée dans la ligne de commande que nous venons de décrire. Il faut tout d'abord savoir que cet en-tête est optionnel. D'ailleurs, une simple requête ne contenant qu'une commande HTTP et une URL est parfaitement utilisable.

Sa structure est la suivante : chaque ligne de l'en-tête comporte un nom de champ (*field name*), suivi du caractère ":" et d'une valeur (*field value*). Chaque ligne est séparée de la suivante par les caractères CRLF. Voici quelques champs fréquemment utilisés dans les requêtes HTTP :

- Content-Encoding : indique l'encodage MIME utilisé par le client dans la requête courante,
- Content-Length : spécifie la longueur du corps de la requête, en octets,
- Content-Type : indique le type d'encodage MIME utilisé pour coder le corps de la requête. Celui utilisé pour transmettre les données html brutes (qui ne sont que du texte) est: `Content-Type: text/html`
- From : permet d'envoyer au serveur l'adresse E-MAIL définie dans les préférences du navigateur,
- If-Modified-Since : est utilisé pour spécifier une date. Ce champ permet au navigateur de ne demander au serveur l'envoi d'un document que si celui-ci a été modifié depuis cette date.
- User-Agent : permet quant à lui d'indiquer au serveur le nom et la version du navigateur utilisé. Cela peut permettre au serveur d'adapter sa réponse en fonction des caractéristiques du navigateur utilisé.

♦ Après l'en-tête optionnel, la requête peut contenir un **corps**, lui aussi optionnel, comportant un certain nombre d'informations dont le format de codage est précisé dans l'en-tête que nous venons de décrire. Le corps ou *body*, séparé de l'en-tête par une ligne vide, n'est en réalité utilisé que lorsqu'on envoie une requête de type POST.

exemple de requête :

```
GET /index.html HTTP/1.0
If-Modified-Since : Sunday, 11-May-1997 19:33:11 GMT
User-Agent : Mozilla/3.0 (WinNT)
```

Ici, on demande l'envoi de la page index.html du serveur sur lequel on est connecté à condition que cette page ait été modifiée depuis le 11 Mai 1997. De plus le client transmet des informations concernant son navigateur.

5.2.3 Réponse HTTP

Etudions maintenant la structure d'une réponse HTTP :

ligne de statut
en-tête
[ligne vide]
corps

- La *ligne de statut* d'une réponse HTTP comprend trois champs : *version code_réponse texte_réponse.*

  - Le premier de ces champs est, comme pour une requête, la version du protocole HTTP utilisé.
  - Le deuxième, *code_réponse*, indique si la requête qui a généré cette réponse a pu être traitée correctement par le serveur.

Code Réponse	Signification
10x	Messages d'information. Non utilisé
20x	Messages indiquant que la requête s'est déroulée correctement
200	Requête OK
201	Requête OK. Création d'une nouvelle ressource (commande POST)
202	Requête OK mais traitement en cours
203	Requête OK mais aucune information à renvoyer (corps vide)
30x	Messages spécifiant une redirection
40x	Erreur due au client
50x	Erreur due au serveur

Les codes les plus souvent rencontrés sont : 200, indiquant que la requête s'est déroulée correctement, 304, spécifiant que la page demandée n'a pas été modifiée depuis la dernière consultation et 404, indiquant que la ressource demandée n'existe pas.

  - Le troisième, *texte_réponse* qui correspond à la signification du message ci-dessus est généralement ajouté par le serveur.

- Examinons maintenant **l'en-tête d'une réponse**. La structure de celui-ci est la même que celle d'une requête. Voici quelques champs que l'on peut rencontrer dans l'en-tête d'une réponse, en sachant que certains de ceux que nous avons vus pour une requête restent valables pour l'en-tête d'une réponse :

  - Date : indique la date de génération de la réponse,
  - Expires : spécifie la date d'expiration de la ressource demandée,
  - Location : contient la nouvelle URL associée au document demandé, lors d'une redirection (codes 30x),
  - Server : précise le nom et la version du serveur ayant envoyé la réponse.

Comme pour une requête, le corps de la réponse est séparé de l'en-tête par une ligne vide.

- Le **corps ou body** contient en fait le document demandé. Cela peut être un fichier HTML simple ou un fichier binaire quelconque, dont le type sera précisé dans l'en-tête par le champ Content-Type.

Exemple de réponse :

```
HTTP/1.0 200 OK
Date : Sunday, 11-May-1999 19:33:14 GMT
Server : Apache/1.1
Content-Type : text/html
Content-Lenght : 65
Last-Modified : Sunday, 11-May-1999 10:54:42 GMT

<HTML> <body>
Bienvenue sur  notre site.....
</body> /HTML>
```

Dans cet exemple, le serveur renvoie une page au format HTML, en précisant quelques informations comme la version du logiciel serveur et la date de dernière modification du fichier considéré.

Le langage HTML ne permet pas à lui seul de créer des documents dynamiques ou interactifs capables par exemple d'indiquer la date du jour ou de donner le résultat d'une requête sur une base de données.

La programmation CGI (Common Gateway Interface) a pour but de construire un document HTML dynamique correspondant à la demande spécifique du client. Le document est envoyé au client au fur et à mesure de sa construction par le serveur.

5.2.4 Http et CGI

Le client indique le nom d'un **fichier** à l'aide d'une URL, non pour recevoir son contenu, mais pour demander son **exécution** au serveur. Ce dernier exécute le programme indiqué et renvoie au client la sortie standard de ce programme (c'est à dire ce que l'on aurait obtenu à l'écran en lançant le programme par une ligne de commande).

Pour pouvoir être exécutés à distance, les exécutables CGI doivent être installés sur le serveur dans un répertoire appelé */cgi-bin*. Tout langage ayant une sortie standard peut être utilisé pour écrire des programmes CGI.

L'interface CGI permet au client d'envoyer au serveur des données saisies par l'utilisateur. Ces données constituent des **arguments d'entrée** pour le programme à exécuter. Elles sont codées selon le format suivant:

nom_champ1=valeur1&nom_champ2=valeur2...

où chaque couple (nom du champ, valeur de saisie) est séparé par un '&' et où nom du champ et valeur de saisie sont séparés par un '='.

De plus, cette chaîne est *URL-encodée*: les espaces y sont remplacés par des '+' et les caractères spéciaux par leur valeur hexadécimale sous le format '%xx'.

Exemple: *NOM=de+la+Maisonneuve&PRENOM=Ren%E9*

Une fois la chaîne des données construite, elle est envoyée au serveur selon la méthode GET ou POST de HTTP.

Avec la méthode GET, la syntaxe d'une URL est la suivante:
http://www.site.fr/cgi-bin/nomExecutable?nom=Dupont&pr%E9nom=Ren%E9

Après le mot clef *cgi-bin* on trouve le nom du programme exécutable, puis un point d'interrogation "?" suivi de la liste des couples (nom,valeur) encodée comme nous l'avons vu tout à l'heure.

Si on utilise la méthode POST, l' URL a la syntaxe suivante :
http://www.site.fr/cgi-bin/nomExecutable

et les données sont envoyées dans le corps de la requête HTTP correspondante.

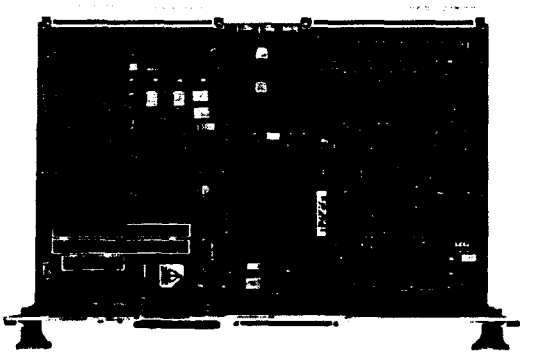
Comparaison :

Avec la méthode GET, les informations sont stockées dans l'URL d'appel du script. Il est alors possible de mémoriser cette URL complète dans un *bookmark* afin de pouvoir exécuter ultérieurement le script avec un certain nombre d'arguments. Cependant, la longueur des données ainsi transférées est limitée. De plus, il est impossible d'envoyer de cette manière des informations sensibles comme un mot de passe, car elles apparaîtraient en clair à l'écran, dans l'URL...

La méthode POST de HTTP résout les deux problèmes évoqués ci-dessus car aucune information n'est passée via l'URL.

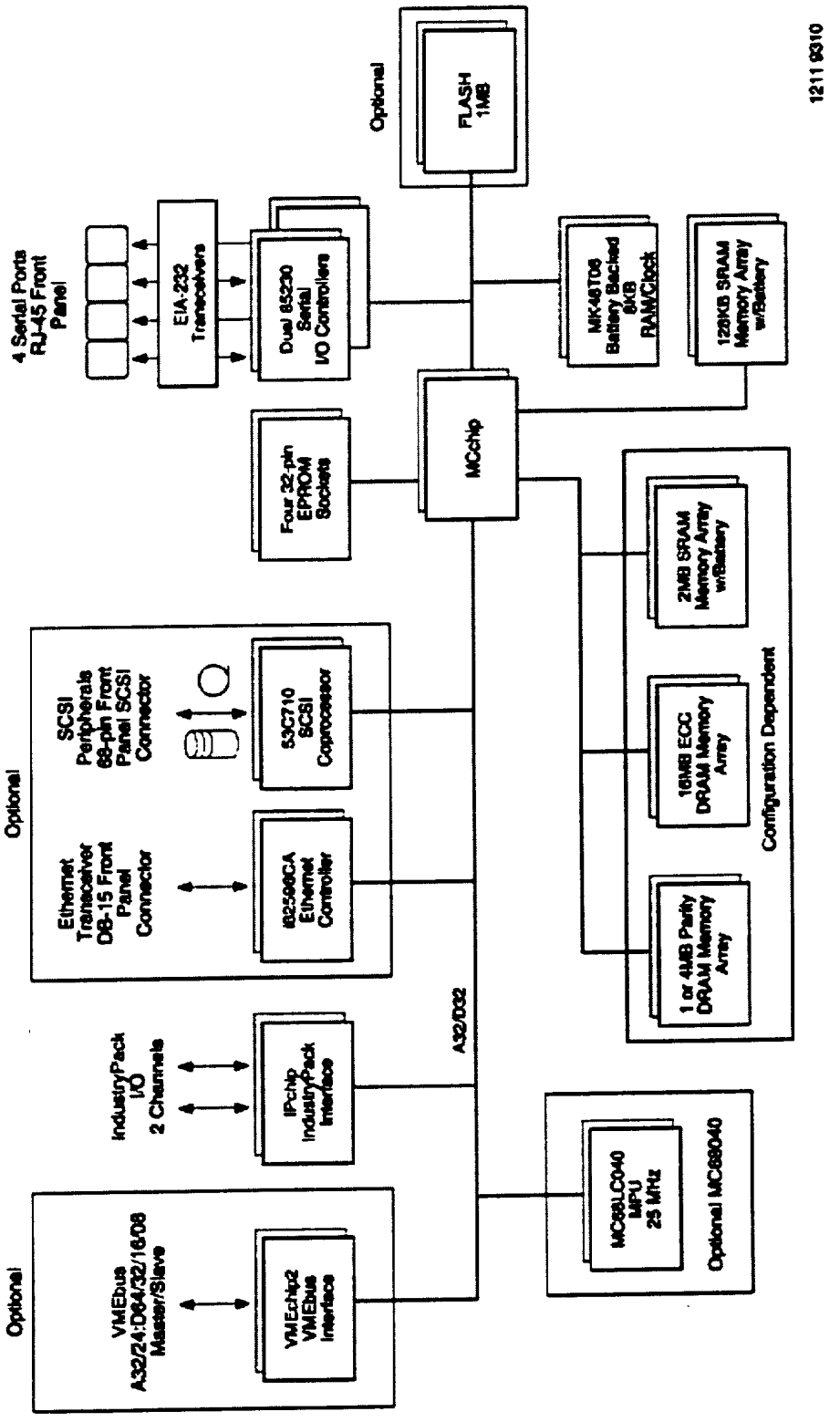
5.3 Carte Lx162

The MVME162LX-222 Embedded Controller is based on the MC68040 microprocessor.



Feature	Description
Microprocessor	MC68040
Memory	4 MB of parity-protected DRAM
	128 KB of SRAM (with battery backup)
	1 MB of Flash memory
	Four EPROM sockets
Real-time clock	8KB NVRAM with RTC and battery backup (SGS-Thomson M48T18)
Switches	RESET and ABORT
Status LEDs	Four: FAIL, RUN, SCON, and FUSES
Tick timers	Four programmable 32-bit timers
Watchdog timer	Provided in MCchip ASIC (VMEchip2)
Serial I/O	EIA-232-D DTE serial interface with four serial ports (Zilog Z85230 controller chips)
SCSI I/O	Optional Small Computer Systems Interface (SCSI) bus interface with 32-bit local bus burst Direct Memory Access (DMA) (NCR 53C710 controller)
Ethernet I/O	LAN Ethernet transceiver interface with 32-bit local bus DMA (Intel 82596CA controller)
IndustryPack Interfaces	2 IndustryPack Interface sites
VMEbus interface	VMEbus requester
	VMEbus system controller
	VMEbus requester
	VMEbus interrupter
	VMEbus interrupt handler
	Eight software interrupts
	Programmable map decoders for the master and slave interfaces
	VMEbus to local bus interface (A24/A32, D8/D16/D32 (D8/D16/D32/D64 BLT) (BLT = Block Transfer) Local-bus-to-VMEbus interface (A16/A24/A32, D8/D16/D32) Two 32-bit programmable Tick Timers and a programmable Watchdog Timer (in the VMEchip2 ASIC) for periodic interrupts
	Global CSR for interprocessor communications
	DMA for fast local memory - VMEbus transfers (A16/A24/A32, D16/D32)

Lx 162 BLOCK DIAGRAM



5.4 IPModule

5.4.1 introduction

Today there is a clear tendency to buy hardware instead of making it. Arguments that go against this tendency are:

- the board with the functions I need is not available;
- the board is available, but has extra functionality I do not need, making it too expensive.

Mezzanine technology may help in solving this problem. Fundamentally it cuts the problem of desired functions in two separate parts. First decide on the processor to be used, then find a mezzanine carrier board with or without system bus interface and finally, personalise the board by adding one or more mezzanines. If the mezzanine board you are looking for is not available, then it is simpler to make a mezzanine board than to make the carrier board.
By buying the carrier and the mezzanine boards or by making some mezzanine boards yourself you will be quicker in the market with a new design and that is vitally important today.

5.4.2 IP-Modules

IP is the abbreviation of "Industry Pack". The standard was prepared by VSO (VITA Standards Organisation) .

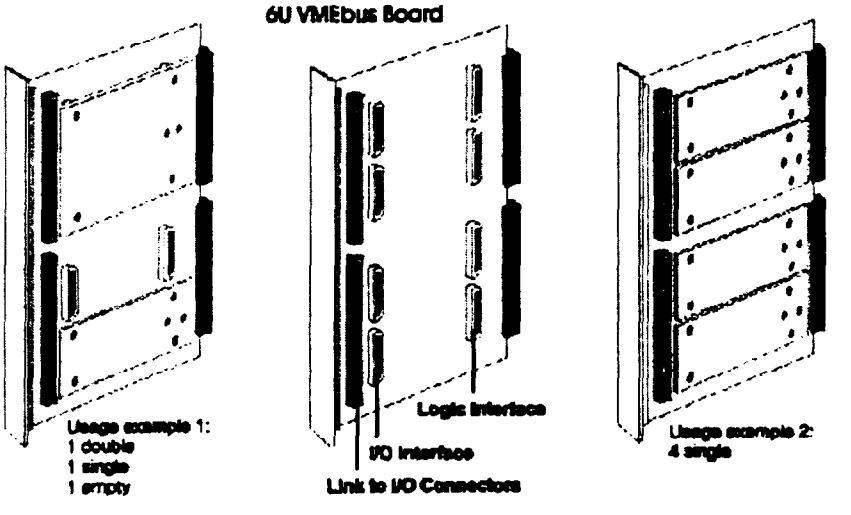
IP modules exist in two sizes: single size and double size, where the double sized IP module can be seen as two single size IP-modules side by side. The single size module measures 1.8 x 3.9 inch (45.3 cm2), the double 3.6 x 3.9 inch (90.6 cm2).

Components on an IP-module can have a maximum height of 0.29 inch (7.4 mm).

The single size IP-module uses two 50-pin connectors: one for interfacing with the carrier board and one to interface with the external world (I/O via the carrier board).

For example, on a 6U VME carrier board there is place for four single size or two double size IP- modules. Remark that theoretically it is possible to put 6 IP modules on a 6U carrier board, but this is not feasible in practice as all I/O lines have to leave via the carrier board and I/F chips are needed.

IP has no front panel solution for the I/O signals



IP Modules on a 6U VMEbus board

5.4.3 Bus description

- The IP-bus is a point to point bus from the carrier board to the IP-module. The IP-module is passive on the bus and can only be accessed by the carrier board. (An IP module is always a slave module).
- There is a maximum of 8 MB addressable memory (*Memory space* mode :22 bit word address [A22 : A1]) and 128 byte I/O space(*IO space mode* : 6 bit word address [A6 :A1]) per single size IP Module.
- In *memory space* mode, the IP-bus has a multiplexed address and data bus (D00 : D15 <==>A7 : A22). This reduces the number of pins needed for the mezzanine board but slows down access.
- Auto configuration (for plug and play) is done via an ID PROM.
- 16 bit data transfers can be used on single size boards and 32 bit transfers can be done on double sized boards. There are byte select lines.
- It is a clocked (or synchronous) bus at 8MHz. 32MHz is optional.
- Two interrupt lines are provided per module.
- Two passive DMA channels are possible. This means that a DMA controller function should be on the carrier board if DMA is needed.

As the IP-bus is simple, it is easy to create custom hardware for it.

Table 1 Signal Identification

Function	Name	No of Pins	Class
Data Bus	D00..D15	16	data
Address	A1..A6	6 (Note 1)	address
Reset	Reset*	1	control
I/O Select	IOSel*	1	control
Memory Select	MemSel*	1	control
Read Int Vector	IntSel*	1	control
Clock	CLK	1	clock
Module Identification	IDSel*	1	control
Data Direction	R/W*	1	control
Data Acknowledge	Ack*	1	control
Byte Select	BS0* BS1*	2	control
Interrupt Request	IntReq0* IntReq1*	2	control
DMA Request	DMAReq0* DMAReq1*	2	control
DMA Acknowledge	DMAck*	1	control
DMA Termination	DMAEnd*	1	control
IP Module Error	Error*	1	control
Function Strobe	Strobe*	1	option
Ground	GND	4	power
+5 Volts	+5V	2	power
+12 Volts	+12V	1	power
-12 Volts	12V	1	power
Reserved		2	reserve
Total		50	

An asterisk (*) after the signal name means that the line is active low. All other signals are active high.

Note 1: During memory access cycles, the 16 data lines are used for the high order address lines (A7..A22).

5.5 Controleur IPIC

This chapter describes the IndustryPack Interface Controller (IPIC) ASIC for the MC68040 bus. The IPIC chip is designed for the MVME162LX board and supports up to two IndustryPack (IP) interfaces, designated IP_a through IP_b.
The IPIC chip converts IP-bound MC68040 read/write/interrupt acknowledge cycles to IndustryPack cycles.

5.5.1 Features

- Provides all logic required to interface the MC68040 bus for two IndustryPacks.
- Supports IndustryPack I/O, Memory, Interrupt Acknowledge, and ID cycles.
- Supports 8-bit, 16-bit, and 32-bit (double size) IndustryPack cycles.
- Provides dynamic bus sizing for accesses to IndustryPack Memory Space.
- Fixed base address for IndustryPack I/O, ID, spaces.
- Programmable base address/size for IndustryPack Memory Space.
- Four Interrupt Handler Control Registers, two for each IndustryPack.
- Recovery timer for each IndustryPack to provide dead time between back to back accesses.

5.5.2 Clock

The clock speed for the IndustryPack logic interface on MVME162LX 200/300 Series boards is 8MHz.

5.5.3 Interrupts

The IPIC chip can be programmed to interrupt the local bus master via the IPL* signal pins when one or more of the four IndustryPack interrupts are asserted. The interrupt control registers allow each interrupt source to be level/edge sensitive and high/low true.
When the local bus master acknowledges an interrupt, if the IPIC determines that it is the source of the interrupt being acknowledged, it waits for IACKIN* to be asserted, then it performs an interrupt acknowledge cycle to the appropriate IndustryPack in order to obtain the vector number. It then passes the vector number on to the local bus master and asserts TA* to terminate the cycle.
When there are multiple IndustryPack interrupts pending at the level being acknowledged, the IPIC performs the interrupt acknowledge for the one with the highest priority. The priority is as follows:

Interrupt Source Pin	Interrupt Source Name	Priority
IntReq0 IP_a	IP_a0	Highest
IntReq1 IP_a	IP_a1	Next Highest
IntReq0 IP_b	IP_b0	Next Lowest
IntReq1 IP_b	IP_b1	Lowest

5.5.4 Memory Map

The following memory map table shows devices selected by the IPIC map decoder.

Address Range	Selected Device	Port Width	Size
$FFF58000-$FFF5807F	IP_a I/O Space	D16	128B
$FFF58080-$FFF580BF	IP_a ID Space	D16	64B
$FFF580C0-$FFF580FF	IP_a ID Space Repeated	D16	64B
$FFF58100-$FFF5817F	IP_b I/O Space	D16	128B
$FFF58180-$FFF581BF	IP_b ID Space	D16	64B
$FFF581C0-$FFF581FF	IP_b ID Space Repeated	D16	64B
$FFFBC000-$FFFBC01F	Control/Status Registers	D32-D8	32B

5.5.5 Programming Model

This section defines the programming model for the control and status registers (CSRs) in the IPIC chip. The possible operations for each bit in the CSR are as follows:

R This bit is a read-only status bit.
R/W This bit is readable and writable.
R/C This status bit is cleared by writing a one to it.
C Writing a zero to this bit clears this bit or another bit. This bit reads as zero.
S Writing a one to this bit sets this bit or another bit. This bit reads as zero.

The possible states of the bits after assertion of the RESET* pin (powerup reset or any local reset) are as defined below.

R The bit is affected by reset.
X The bit is not affected by reset.

The CSR registers (0xFFFBC000-0xFFFBC01F) can be accessed as bytes, words, or longwords. They should not be accessed as lines. They are shown in the table as bytes, and the bits in the following register descriptions are labeled as bits 7 through 0.

Offset	Register Name	Register Bit Names							
		D7	D6	D5	D4	D3	D2	D1	D0
$00	CHIP ID	0	0	1	0	0	0	1	1
$01	CHIP REVISION	0	0	0	0	0	0	0	0
$02	RESERVED	0	0	0	0	0	0	0	0
$03	RESERVED	0	0	0	0	0	0	0	0
$04 -$0F	NOT USED								
$10	IP a INT0 CONTROL	a0_PLTY	a0_E/L*	a0_INT	a0_IEN	a0_ICLR	a0_IL2	a0_IL1	a0_IL0
$11	IP a INT1 CONTROL	a1_PLTY	a1_E/L*	a1_INT	a1_IEN	a1_ICLR	a1_IL2	a1_IL1	a1_IL0
$12	IP b INT0 CONTROL	b0_PLTY	b0_E/L*	b0_INT	b0_IEN	b0_ICLR	b0_IL2	b0_IL1	b0_IL0
$13	IP b INT1 CONTROL	b1_PLTY	b1_E/L*	b1_INT	b1_IEN	b1_ICLR	b1_IL2	b1_IL1	b1_IL0
$14 -$17	NOT USED								
$18	IP a GENERALCONTROL	a_ERR	0	a_RT1	a_RT0	a_WIDTH1	a_WIDTH0	0	a_MEN
$19	IP b GENERALCONTROL	b_ERR	0	b_RT1	b_RT0	b_WIDTH1	b_WIDTH0	0	b_MEN
$1A -$1B	NOT USED								
$1C	RESERVED	0	0	0	0	0	0	0	0
$1D	RESERVED	0	0	0	0	0	0	0	0
$1E	RESERVED	0	0	0	0	0	0	0	0
$1F	IP RESET	0	0	0	0	0	0	0	RES

Chip ID Register

The read-only Chip ID Register is hard-wired to a hexadecimal value of $23. Writes to this register do nothing, however the IPIC terminates them normally with TA*.

ADR/SIZ	$FFFBC000 (8 bits)							
BIT	7	6	5	4	3	2	1	0
NAME	CID7	CID6	CID5	CID4	CID3	CID2	CID1	CID0
OPER	R	R	R	R	R	R	R	R
RESET	0	0	1	0	0	0	1	1

Chip Revision Register

The read-only Chip Revision Register is hard-wired to reflect the revision level of the IPIC ASIC. The current value of this register is $00. Writes to this register do nothing, however the IPIC terminates them normally with TA*.

ADR/SIZ	$FFFBC001 (8 bits)							
BIT	7	6	5	4	3	2	1	0
NAME	REV7	REV6	REV5	REV4	REV3	REV2	REV1	REV0
OPER	R	R	R	R	R	R	R	R
RESET	0	0	0	0	0	0	0	0

IP_a, IP_b : IRQ0/IRQ1 Interrupt Control Registers

ADR/SIZ	$FFFBC010 through $FFFBC013 (8 bits each)							
BIT	7	6	5	4	3	2	1	0
NAME($10)	a0_PLTY	a0_E/L*	a0_INT	a0_IEN	a0_ICLR	a0_IL2	a0_IL1	a0_IL0
NAME($11)	a1_PLTY	a1_E/L*	a1_INT	a1_IEN	a1_ICLR	a1_IL2	a1_IL1	a1_IL0
NAME($12)	b0_PLTY	b0_E/L*	b0_INT	b0_IEN	b0_ICLR	b0_IL2	b0_IL1	b0_IL0
NAME($13)	b1_PLTY	b1_E/L*	b1_INT	b1_IEN	b1_ICLR	b1_IL2	b1_IL1	b1_IL0
OPER	R/W	R/W	R	R/W	C	R/W	R/W	R/W
RESET	0 R	0 R	0 R	0 R	0 R	0 R	0 R	0 R

IL2-IL0
These three bits select the interrupt level for the corresponding IndustryPack interrupt request. Level 0 does not generate an interrupt.

ICLR
In edge-sensitive mode, writing a logic 1 to this bit clears the corresponding INT status bit. In level-sensitive mode, this bit has no function. It always reads as 0.

IEN
When IEN is set, the interrupt is enabled. When IEN is cleared, the interrupt is disabled.

INT
When this bit is high, an interrupt is being generated for the corresponding IndustryPack IRQ. The interrupt is at the level programmed in IL2-IL0.

E/L*
When this bit is high, the interrupt is edge sensitive. When the bit is low, the interrupt is level sensitive.

PLTY
When this bit is low, interrupt is activated by a falling edge/low level of the IndustryPack IRQ*. When this bit is high, interrupt is activated by a rising edge/high level of the IndustryPack IRQ*. Note that if this bit is changed while the E/L* bit is set (or is being set), an interrupt may be generated. This can be avoided by setting the ICLR bit during write cycles that change the PLTY bit. Because IndustryPack IRQ*s are active low, PLTY would normally be cleared.

IP_a, IP_b: General Control Registers

ADR/SIZ	$FFFBC018 through $FFFBC019 (8 bits each)							
BIT	7	6	5	4	3	2	1	0
NAME($18)	a_ERR	0	a_RT1	a_RT0	a_WIDTH1	a_WIDTH0	0	a_MEN
NAME($19)	b_ERR	0	b_RT1	b_RT0	b_WIDTH1	b_WIDTH0	0	b_MEN
OPER	R	R	R/W	R/W	R/W	R/W	R	R/W
RESET	? R	0 R	0 R	0 R	0 R	0 R	0 R	0 R

MEN
a_MEN / b_MEN enable the local bus to perform read/write accesses to their corresponding IndustryPack memory space when set, and disable such accesses when cleared

WIDTH1, WIDTH0
The IPIC assumes the memory space data-bus width of IP_a and IP_b to be the value decoded from its control bits WIDTH1 and WIDTH0. The following table shows widths inferred by these bits.

WIDTH1	WIDTH0	Memory Space Data Width
0	0	32 bits
0	1	8 bits
1	0	16 bits
1	1	reserved

RT1,RT0
The recovery-timers determine the time that must expire from the acknowledgment of an IndustryPack I/O, ID, or Interrupt Acknowledge cycle until the IPIC asserts a new I/O, ID, or Int SEL* to the same IndustryPack.

RT1	RT0	Recovery Time
0	0	0 microseconds
0	1	2 microseconds
1	0	4 microseconds
1	1	8 microseconds

ERR
When one of these bits is set to a one, its corresponding IndustryPack Error* signal is asserted.

IP RESET Register

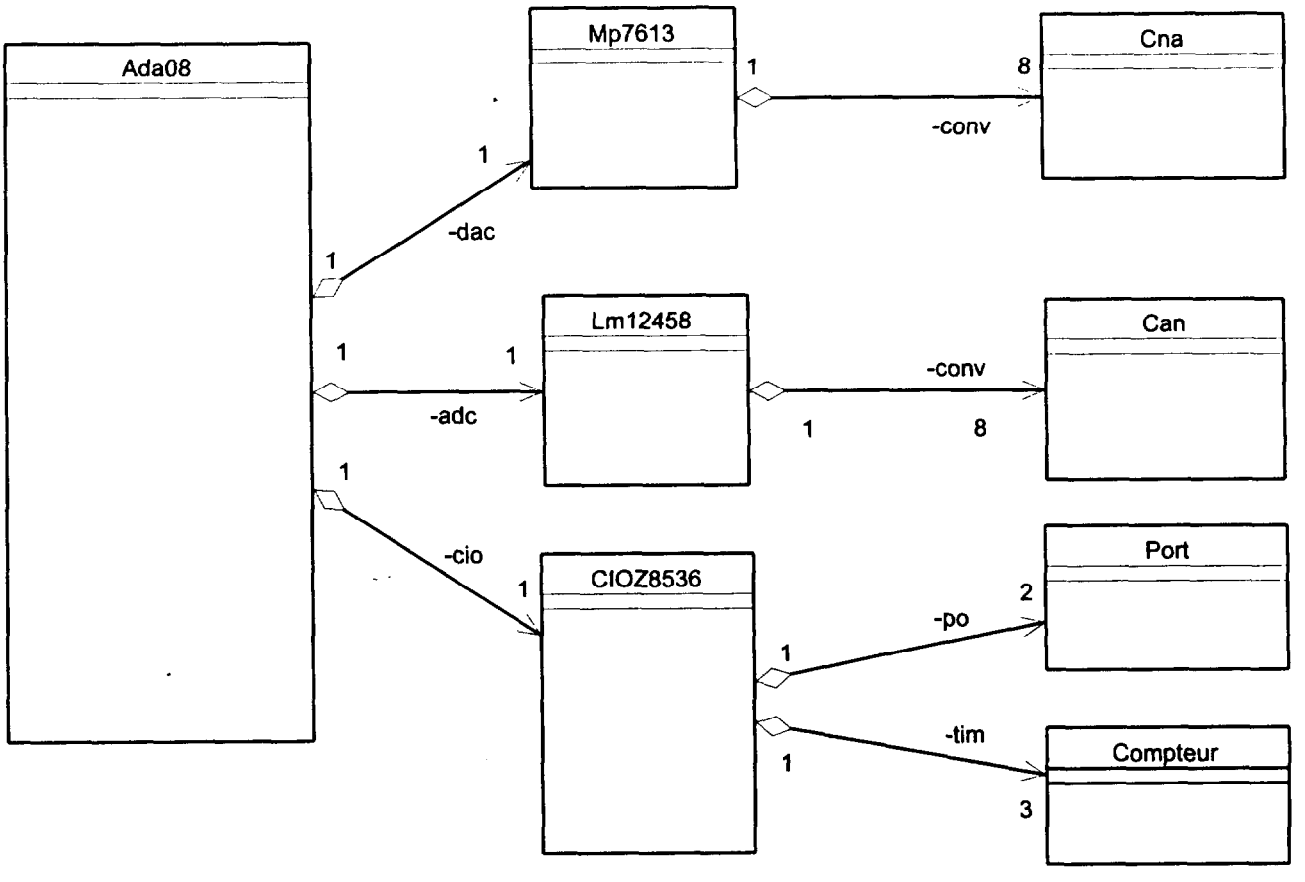
ADR/SIZ	$FFFBC01F (8 bits)							
BIT	7	6	5	4	3	2	1	0
NAME	0	0	0	0	0	0	0	RES
OPER	R	R	R	R	R	R	R	S
RESET	0 R	0 R	0 R	0 R	0 R	0 R	0 R	0 R

RES
Setting RES to a one asserts the IPIC IPRESET* signal.

5.6 Module ADA08

Le module ADA 08 de ACTIS Computer permet de réaliser de nombreuses fonctions avec un simple module IP:

1. Système d'acquisition de données 12-bit + signe avec 8 entrées unipolaires ou 4 entrées différentielles.
2. Huit convertisseurs numériques analogiques 12 bits.
3. Deux ports 8 bits avec 4 lignes de « *handshake* ».
4. Trois temporisateurs 16 bits.



Nota : Les pages suivantes constituent la documentation constructeur de ce module. Les notices des composants MP7613, LM12458 et CIO Z8536 ne sont pas nécessaires pour répondre aux questions du sujet d'examen.