//readme 104033112 廖彥綸
//iCNC 的架構

Form 藉由 MMI 進入 iCNC 的主程式，流程為：
Form 呼叫 CNC ->指標 cncDataPtr 指向 MMI-opMode 中的 MEM_MODE，目前只作出簡易的連結，另外還有 JOG_MODE，為連結於程式中。
進入 MMI 後也等同於正式進入 iCNC。
cncDataPtr->mmiData.opMode = MEM_MODE;

進入 iCNC 後，MMI 將訊息傳給程式其他部分，以專業分工讓各個區塊處理專門的工作。而 MMI 本身是處理人和機器介面間的各種互動，包含以下函式：
mmiInit -> 初始化mmi 內部的資料，避免讀取到舊資料而影響程式的情形。其中包括呼叫 clearByteSpace 清除 MMI_DATA 的內部資料、將 FileName 和 Coordinate 設定成 '\0' (空字元)，和重置多個 bool值。在 mmi.h 內預先宣告會使用到的函式，而函式定義的地方設定在 mmi function，在讓 mmi.main 和 mmi function include mmi.h，避免程式因為函式的宣告順序影響而發生錯誤。

mmiIdle -> 這個函式用來確認介面選擇的模式(MEM_MODE 或 JOG_MODE)，並將資料傳給 link，由 link做個資料的傳遞，使他被做有效率的分析。

mmiMemState 和 mmiJogState -> 先判斷是否已經有資料在傳輸，若資料正在傳輸，會設定 ncFileValid 為 true 並阻止下一筆資料的進入(以免發生問題)，當資料傳輸完畢會將 ncFileValid 重新設定為 false 此時代表資料已經接收完畢，內部已經沒有帶傳輸的資料，所以允許下一筆資料的進入。由於目前只對 Form 做簡單的連結，所以只開啟 mmiMemState。

mmiReset -> 檔案傳輸完畢後，對檔案內部做再一次的淨空。

mmiMain -> 對 mmiDataPtr 指向的 mmiState 會在 mmiMain 做第一步的判斷，並使用 switch - case 迴圈呼叫以上各個函式。

資料傳入 MMI 後會再 mmiIdle 將資料傳輸到 link。link 就像是個資料間的橋樑，負責做傳輸資料間的訊息的工作，內部的函式皆與此有關；包含MMI、DEC 和 INTP之間的流通，皆以 LINK作為媒介。命名時以 LINK 和另一資料的名稱作為命名的標準，如此可增加辨別資料類別的容易度及清晰度。函式定義的地方銅像設定在 link.h 中，以 link.main include link.h 的方式讓函式定義與宣告分離，方便程式撰寫。

linkInit -> 進入 link 後優先將 link 中的內部資料淨空，之前傳輸過的資料遺留在檔案中。在此步驟，會利用指標的方法(linkDataPtr)呼叫各個函式的 Init ，包括mmiToLinkDataInit、linkToMmiDataInit、decToLinkDataInit、linkToDecDataInit、intpToLinkDataInit、linkToIntpDataInit ，並獨自在函式內進行初始化，以及各 bool 的 false 值設定。

MMI_TO_LINK_DATA、LINK_TO_MMI_DATA -> 負責 MMI 和 LINK 之間的資料流動，其中包含的資料除了本身需要用的的資料變數，還宣告了該函式的 bool 以判斷此資料是否已經過傳輸。若資料已經傳輸，則設定 Vaild 值為 true ，在需一利用這些判別時，則可輕易的呼叫。

DEC_TO_LINK_BLOCK、LINK_TO_DEC_DATA -> 執行 DEC 和 LINK 的聯絡，在 DEC_TO_LINK_BLOCK 中輸出 DEC 以解碼的資料至 LINK 作為最後的打包公用。輸入方面，也利用 LINK_TO_DEC_DATA 輸入資料至 DEC 做程式最主要的解碼工作。

INTP_TO_LINK_DATA、LINK_TO_INTP_DATA -> 負責 LINK 和 INTP 的資料傳遞，但 INTP 尚未完成，所以此項先擱置在一邊。

DEC 負責程式最主要的解碼工作，在 dec function 中有 spaceFunction、nFunction、gFunction、mFunction、fFunction、xyFunction，處理輸入文字文件中的 '空格'、'N'、'G'、'M'、'F'、'X'、'Y' 字元開頭的 CNC 碼，開頭出現這些字元後，將指標指向各自的函式，根據函式內部的規則做解碼的動作，當輸入的資料不符合 CNC 的規則時我們必須將程式指向 ERROR 表示程式已經出錯，不需要再進行之後的動作，而當輸入的 CNC 碼皆符合規定，則會依照各 Function 內部的程式做程式碼的解析。另外當解碼的結果為 '\0'、'\n' 和我們自己定義的 ';' 時會被認定為一段 CNC 碼的結束，除了呼叫 Init 重置各資料及 bool ，還會呼叫 endOfLine 及 DEC_WRITE_OUT，做寫出的預備。

decInit -> 進入 dec 後先由 decInit 淨空所以資料。(markG90G91 擁有記憶性，意指第一次輸入 G90或G91 後可以不重複輸入，除非想改變 G90G91 的型態，所以不清除 markG90G91 的值)

decIdle -> 從 LINK 接收檔案名稱，其中充分條件為 link 有檔案傳給 dec 以 linkToDecDataValid 做判斷，必要條件為 1. 檔案已成功接收，以 ncFileValid 判斷是否已經有檔案在 dec 中，若其值為 true 則表示檔案已接收，若接收成功則設定 ncFileValid 為 false ，允許接收下個檔案；2. START 按鈕被按下。如果

以上條件皆達成，則檔案名稱將成功複製 (利用strcpy)，並且將 DEC 指向下個步驟，呼叫 DEC_OPEC_NC_FILE。

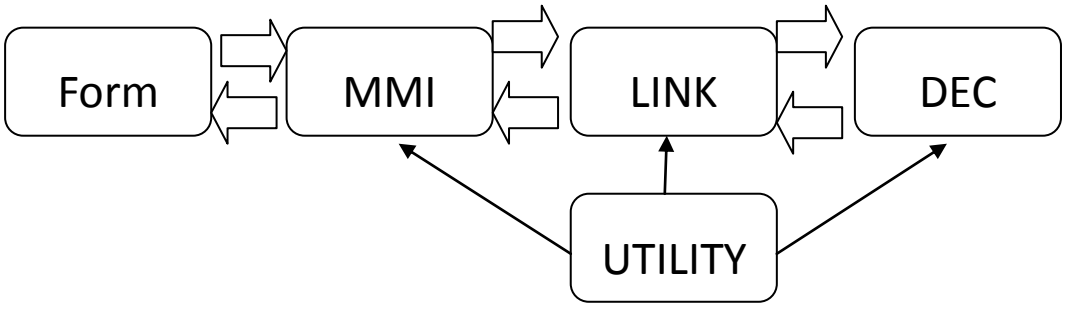
decOpenNcFile -> 由於無法預知進入的檔案大小，所以利用動態記憶體讀取檔案內容。撰寫判斷式，若進入的檔案式空檔案或是不合法的位置 (NULL)，則是發生錯誤，即讓 DecDataPtr -> decState = DEC_ERROR。若輸入的資料 !=NULL 時程式則會被執行，執行成功後 DecDataPtr 會指向下一個步驟， DEC_LOOP。

decLoop -> loop 是執行解碼的主要部分，依照讀取的字元呼叫不同的函式，可接受的字元為 '空格'、'N'、'G'、'M'、'F'、'X'、'Y' 等..... 也可以依照程式設計師的需求定義需要的字元，如定義 '；' 為結束字元。如果出現了未定義的字元，為了保護程式不讓他產生預料外的結果，我們將 DecDataPtr -> decState = DEC_ERROR ，而各個 Function的解碼規則由 Function 定義。如： NN100 、N100 X0.5 Y1.0 被視為不合法的語法，前者重複了字元 N，後者為符合 CNC的規定 (遺失 G00碼)。總之，若輸入的資料有誤，則需要做 DecDataPtr -> decState = DEC_ERROR 強制結束 DEC_LOOP 階段。若資料解碼完畢，且未有任何錯誤產生，在結束字元 '\0'、'\n'、'；' 的地方會將 DecDataPtr 導向 DEC_WRITE_OUT。

decWriteOut -> 是寫出的程式，和 linkToDecData 相反，他將資料傳輸至 decToLinkData，同之前做的條件判斷，必須通過 decToLinkDataValid 的 true 、false 判別，如果做後出現了 M30 ; (M30被認為是 CNC code 最後的結尾)所以資料會被送到 decWaitProgramEnd 。如果結尾沒有 M30 表示這只是一個分行符號，資料從下一個開頭處回傳至 decIdle 。

decWaitProgramEnd -> 這個函式是用來淨空動態記憶體，利用 free() 釋放 Ptr 指標指向的記憶體空間，以便節省記憶體。如果省略了這個步驟，記憶體的配置會成為一個問題，也有可能讓不必要的資料佔據記憶體，導致記憶體不夠用而發生錯誤。

對於各區域間之關係，可以用流程圖得到更清楚的說明。



UTILITY 做為通用函式庫，將常用的函式放在 utility.main ，可以支援各個函式庫；MMI 連結 Form 和 iCNC 的內部程式，LINK 彙整所有工作區，讓 iCNC有效率整理串聯的資訊。

遇到問題與解答：

1. 每個人對專案的想法都不盡相同，包括老師、助教、及所有的同學。我記錄老師上課時所撰寫的函式，和助教演習課講解的內容，設定的變數不一定相同，所以演習課教導的程式碼如果直接貼上可能出現許多 error，解決方法是自己對程式必須有一定程度的理解，然後實際執行一遍。如此才能對自己程式碼的位置、名稱、內容有實質的了解。

2. debug 時常出現許多錯誤，此時會先看 C 給我們的錯誤形式，大部分的錯誤是為宣告變數、函式、標頭檔，當完成宣告之後問題即可解決；若問題類行為..... is not a member of .....，可能問題和第一點有所關連，因為每個人對程式的宣告不同，可能是資料的名稱 (包括大小寫差異)而造成的 error 。所以程式要統一資料型態、名稱、宣告，至少要自己認可的宣告方法。另外，在資料結構中只表的資料結構利用 -> ，而其他結構體的指向用 .，向一般常見的跳出視窗結構可能為 System.inform(" ") 我們是用 . 做 System 中 inform 的呼叫。但在此專案中，我們常將結構體用指標 (或包含在另一結構體中) 做指向，所以呼叫時常利用 -> 的方法。舉例來說，我們在 LINK 中的 mmiToLinkDataInit：利用 linkDataPtr 指向內部的 mmiToLinkData ，根據宣告：

```
LINK_DATA        linkData;
LINK_DATA        *linkDataPtr = &linkData;
```

linkDataPtr 明顯是指標，所以在宣告時用 -> 指向下一個結構體，但在程式中，mmiToLinkDataInit並不是一個指標性的結構體，所以用 . 的方式呼叫內部資料，

```
linkDataPtr->mmiToLinkData.ncFileName[i] = '\0';
```

錯誤也有可能是結構體呼叫方式弄錯而產生，或者是宣告的知料在結構中但未準確執行呼叫的動作。最後一個方法是利用錯誤的代號尋找資料，debug 時底下可能會出現 error C2065 之類的代號，用 Google 的方式查詢可以找到許多有相同問題的人，參考他們的解答方法對自己解決問題的方法大有助益。最後不要害怕 ERROR 的數量，也許幾十個 ERROR 都來自於函式宣告時遺漏了一個 ';'。

3. 出現了真的不明白的問題可以嘗試和同學或助教討論，當然不會直接複製別人的程式嗎，是聽聽看別人的想法，並對程式坐進一步的解讀。或請助教幫忙指點迷經，助教的能力比我們高，也許可以快速的發現問題的癥結點。

4. 進行程式時會出現許多預料之外的問題，第一次 Debug 時，發現程式進入 DEC_IDLE 後即在 DEC_IDLE 無限重複，請問助教後才發現程式碼的問題，我原

本該在 START 中設定一個布林值為 true，而卻在 load 檔案後即設定他為 true。而導致按下 START 後參數不會進行轉換，產生後續的問題，學習玩 Debug 的流程後，我也擁有判斷程式碼邏輯性的能力，但如要修改正確的程式碼還需要多學習。